Programming Environment HP-UX Concepts and Tutorials

HP Part Number 97089-90042



Hewlett-Packard Company
3404 East Harmony Road, Fort Collins, Colorado 80525

NOTICE

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MANUAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

WARRANTY

A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

Copyright © Hewlett-Packard Company 1986

This document contains information which is protected by copyright. All rights are reserved. Reproduction, adaptation, or translation without prior written premission is prohibited, except as allowed under the copyright laws.

Restricted Rights Legend

Use, duplication or disclosure by the U.S. Government Department of Defense is subject to restrictions as set forth in paragraph (b)(3)(ii) of the Rights in Technical Data and Software clause in FAR 52.227-7013.

Copyright © AT&T, Inc. 1980, 1984

Copyright © The Regents of the University of California 1979, 1980, 1983

This software and documentation is based in part on the Fourth Berkeley Software Distribution under license from the Regents of the University of California.

Printing History

New editions of this manual will incorporate all material updated since the previous edition. Update packages may be issued between editions and contain replacement and additional pages to be merged into the manual by the user. Each updated page will be indicated by a revision date at the bottom of the page. A vertical bar in the margin indicates the changes on each page. Note that pages which are rearranged due to changes on a previous page are not considered revised.

The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections and updates which are incorporated at reprint do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

August 1986...Edition 1

Table of Contents

HP-UX Programming

Introduction 1
Basics 2
Program Arguments 2
The "Standard Input" and "Standard Output" 3
The Standard I/O Library 5
File Access 5
Error Handling – Stderr and Exit 8
Miscellaneous I/O Functions 9
Low-Level I/O 10
File Descriptors 10
Read and Write 11
Open, Close, Unlink 13
Random Access – Lseek 15
Error Processing 16
Processes 17
The "System" Function 17
Low-Level Process Creation – Execl and Execv 17
Control of Processes – Fork and Wait 19
Pipes 20
Signals – Interrupts and All That 23
Appendix – The Standard I/O Library 28
General Usage 28
Calls 29

HP-UX Programming

Introduction

This tutorial describes how to write programs that interface with the HP-UX operating system in a non-trivial way. This includes programs that use files by name, that use pipes, that invoke other commands as they run, or that attempt to catch interrupts and other signals during execution.

The document collects material which is scattered throughout several sections of the *HP-UX Reference* manual. There is no attempt to be complete; only generally useful material is dealt with. It is assumed that you will be programming in C, so you must be able to read the language roughly up to the level of *The C Programming Language*. Some of the material in this tutorial is based on topics covered more carefully there. You should also be familiar with HP-UX itself.

Basics

Program Arguments

When a C program is run as a command, the arguments on the command line are made available to the function *main* as an argument count *argc* and an array *argv* of pointers to character strings that contain the arguments. By convention, *argv[0]* is the command name itself, so *argc* is almost always[1] greater than 0.

The following program illustrates the mechanism: it simply echoes its arguments back to the terminal. (This is essentially the *echo* command.)

```
main(argc, argv)          /* echo arguments */
int argc;
char *argv[];
{
    int i;

    for (i = 1; i < argc; i++)
        printf("%s%c", argv[i], (i<argc-1) ? ' ' : '\n');
}
```

argv is a pointer to an array whose individual elements are pointers to arrays of characters; each is terminated by \0, so they can be treated as strings. The program starts by printing *argv[1]* and loops until it has printed them all.

The argument count and the arguments are parameters to *main*. If you want to keep them around so other routines can get at them, you must handle them like any other argument you want to pass on.

[1] Direct calls to *exec*(2) could violate this condition. Programs that use *argv[0]* usually assume that it is present, but this improper invocation could cause strange failures.

The "Standard Input" and "Standard Output"

The simplest input mechanism is to read the "standard input", which is generally the user's terminal. The function *getchar* returns the next input character each time it is called. A file can be substituted for the terminal by using the < convention: if *prog* uses *getchar*, then the command line

```
prog <file
```

causes *prog* to read *file* instead of the terminal. *Prog* itself need know nothing about where its input is coming from. This is also true if the input comes from another program via the HP-UX pipe mechanism:

```
otherprog | prog
```

provides the standard input for *prog* from the standard output of *otherprog*.

Getchar returns the value *EOF* when it encounters the end-of-file (or an error) on whatever you are reading. The value of *EOF* is normally defined to be *-1*, but it is unwise to take any advantage of that knowledge. As will become clear shortly, this value is automatically defined for you when you compile a program, and need not be of any concern.

Similarly, *putchar(c)* puts the character *c* on the "standard output", which is also by default the terminal. The output can be captured on a file by using >: if *prog* uses *putchar*,

```
prog >outfile
```

writes the standard output on *outfile* instead of the terminal. *outfile* is created if it doesn't exist; if it already exists, its previous contents are overwritten. And a pipe can be used:

```
prog | otherprog
```

puts the standard output of *prog* into the standard input of *otherprog*.

The function *printf*, which formats output in various ways, uses the same mechanism as *putchar* does, so calls to *printf* and *putchar* can be intermixed in any order; the output will appear in the order of the calls.

Similarly, the function *scanf* provides for formatted input conversion; it will read the standard input and break it up into strings, numbers, etc., as desired. *scanf* uses the same mechanism as *getchar*, so calls to them can also be intermixed.

Many programs read only one input and write one output; for such programs I/O with *getchar*, *putchar*, *scanf*, and *printf* may be entirely adequate, and it is almost always enough to get started. This is particularly true if the HP-UX pipe facility is used to connect the output of one program to the input of the next. For example, the following program strips out all ASCII control characters from its input (except for newline and tab).

```
#include <stdio.h>

main()        /* ccstrip: strip non-graphic characters */
{
    int c;
    while ((c = getchar(\|)) != EOF)
        if ((c >= ' ' && c < 0177) || c == '\t' || c == '\n')
            putchar(c);
    exit(0);
}
```

The line

```
#include <stdio.h>
```

should appear at the beginning of each source file. It causes the C compiler to read a file (*/usr/include/stdio.h*) of standard routines and symbols that includes the definition of *EOF*.

If it is necessary to treat multiple files, you can use *cat* to collect the files for you:

```
cat file1 file2 . . . | ccstrip >output
```

and thus avoid learning how to access files from a program. By the way, the call to *exit* at the end is not necessary to make the program work properly, but it assures that any caller of the program will see a normal termination status (conventionally 0) from the program when it completes. Section 6 discusses status returns in more detail.

The Standard I/O Library

The "Standard I/O Library is a collection of routines intended to provide efficient and portable I/O services for most C programs. The standard I/O library is available on each system that supports C, so programs that confine their system interactions to its facilities can be transported from one system to another essentially without change.

In this section, we will discuss the basics of the standard I/O library. The appendix contains a more complete description of its capabilities.

File Access

The programs written so far have all read the standard input and written the standard output, which we have assumed are magically pre-defined. The next step is to write a program that accesses a file that is **not** already connected to the program. One simple example is *wc*, which counts the lines, words and characters in a set of files. For instance, the command

```
wc x.c y.c
```

prints the number of lines, words and characters in *x.c* and *y.c* and the totals.

The question is how to arrange for the named files to be read – that is, how to connect the file system names to the I/O statements that actually read the data.

The rules are simple. Before it can be read or written a file has to be **opened** by the standard library function *fopen*. *Fopen* takes an external name (like *x.c* or *y.c*), does some housekeeping and negotiation with the operating system, and returns an internal name which must be used in subsequent reads or writes of the file.

This internal name is actually a pointer, called a *file pointer*, to a structure which contains information about the file, such as the location of a buffer, the current character position in the buffer, whether the file is being read or written, and the like. Users don't need to know the details, because part of the standard I/O definitions obtained by including *stdio.h* is a structure definition called `FILE`. The only declaration needed for a file pointer is a line resembling:

```
FILE *fp, *fopen();
```

This says that *fp* is a pointer to a `FILE`, and *fopen* returns a pointer to a `FILE` (`FILE` is a type name, like `int`; not a structure tag).

The actual call to *fopen* in a program is

```
fp = fopen(<name>, <mode>);
```

The first argument of *fopen* is the <name> of the file, as a character string. The second argument is the <mode>, also as a character string, which indicates how you intend to use the file. The only allowable modes are read (`r`) write (`w`) or append (`a`), and their updating counterparts (`r+`, `w+`, and `a+`).

If you open a non-existent file for writing or appending, it is created (if possible). Opening an existing file for writing causes the old contents to be discarded. Trying to read a file that does not exist is an error, and there may be other causes of error as well (like trying to read a file when you don't have permission). If there is any error, *fopen* will return the null pointer value `NULL` (defined in *stdio.h*).

The next thing needed is a way to read or write the file once it is open. There are several possibilities, of which *getc* and *putc* are the simplest. *getc* returns the next character from a file; it needs the file pointer to tell it what file. Thus

```
c = getc(fp)
```

places in *c* the next character from the file referred to by *fp*; it returns *EOF* when it reaches end of file. *putc* is the inverse of *getc*:

```
putc(c, fp)
```

puts the character *c* on the file *fp* and returns *c*. *Getc* and *putc* return EOF on error.

When a program is started, three files are opened automatically, and file pointers are provided for them. These files are the standard input, the standard output, and the standard error output; the corresponding file pointers are called *stdin*, *stdout*, and *stderr*. Normally these are all connected to the terminal, but can be redirected to files or pipes as described in the Basics section earlier in this tutorial. *Stdin*, *stdout* and *stderr* are pre-defined in the I/O library as the standard input, output and error files; they can be used anywhere an object of type `FILE *` can be. They are constants, however, **not** variables, so don't try to assign to them.

With some of the preliminaries out of the way, we can now write *wc*. The basic design is one that has been found convenient for many programs: if there are command-line arguments, they are processed in order. If there are no arguments, the standard input is processed. This way the program can be used stand-alone or as part of a larger process.

```
#include <stdio.h>

main(argc, argv)        /* wc: count lines, words, chars */
int argc;
char *argv[];
{
    int c, i, inword;
    FILE *fp, *fopen();
    long linect, wordct, charct;
    long tlinect = 0, twordct = 0, tcharct = 0;

    i = 1;
    fp = stdin;
    do {
        if (argc > 1 && (fp=fopen(argv[i], "r")) == NULL) {
            fprintf(stderr, "wc: can't open %s\n", argv[i]);
            continue;
        }
        linect = wordct = charct = inword = 0;
        while ((c = getc(fp)) != EOF) {
            charct++;
            if (c == '\n')
                linect++;
            if (c == ' ' || c == '\t' || c == '\n')
                inword = 0;
            else if (inword == 0) {
                inword = 1;
                wordct++;
            }
        }
        printf("%7ld %7ld %7ld", linect, wordct, charct);
        printf(argc > 1 ? " %s\n" : "\n", argv[i]);
        fclose(fp);
        tlinect += linect;
        twordct += wordct;
        tcharct += charct;
    } while (++i < argc);
    if (argc > 2)
        printf("%7ld %7ld %7ld total\n", tlinect, twordct, tcharct);
    exit(0);
}
```

The function *fprintf* is identical to *printf* except that the first argument is a file pointer that specifies the file to be written.

The function *fclose* is the inverse of *fopen*; it breaks the connection between the file pointer and the external name that was established by *fopen*, freeing the file pointer for another file. Since there is a limit on the number of files that a program can have open simultaneously, it's a good idea to release resources when they are no longer needed. There is also another reason to call *fclose* on an output file — it flushes the buffer in which *putc* is collecting output (*fclose* is called automatically for each open file when a program terminates normally).

Error Handling – Stderr and Exit

Stderr is assigned to a program in the same way that *stdin* and *stdout* are. Output written on *stderr* appears on the user's terminal even if the standard output is redirected. *Wc* writes its diagnostics on *stderr* instead of *stdout* so that if one of the files can't be accessed for some reason, the message finds its way to the user's terminal instead of disappearing down a pipeline or into an output file.

The program actually signals errors in another way, using the function *exit* to terminate program execution. The argument of *exit* is available to whatever process called it, so the success or failure of a program can be tested by another program that uses it as a sub-process. By convention, a return value of 0 signals that all is well; non-zero values signal abnormal situations. The preceding example, *wc*, has only a one exit condition, so it provides no means for detecting errors when it is used as a sub-process.

Exit itself calls *fclose* for each open output file, to flush out any buffered output, then calls a routine named *_exit*. The function *_exit* causes immediate termination without any buffer flushing; it can be called directly if desired. Use of *_exit* becomes necessary when terminating a parent and child process because both processes set up variables and buffers that are duplicates of each other. If *_exit* is not used during termination of at least one of the processes, both sets of buffers are flushed, causing duplicate output.

Miscellaneous I/O Functions

The standard I/O library provides several other I/O functions besides those previously illustrated.

Normally output with *putc*, etc., is buffered (except to *stderr*); to force it out immediately, use *fflush(fp)*.

Fscanf is identical to *scanf*, except that its first argument is a file pointer (as with *fprintf*) that specifies the file from which the input comes; it returns EOF at end of file.

The functions *sscanf* and *sprintf* are identical to *fscanf* and *fprintf*, except that the first argument names a character string instead of a file pointer. The conversion is done from the string for *sscanf* and into it for *sprintf*.

fgets(buf, size, fp) copies the next line from *fp*, up to and including a newline, into *buf*; at most *size-1* characters are copied; it returns NULL at end of file. *fputs(buf, fp)* writes the string in *buf* onto file *fp*.

The function *ungetc(c, fp)* "pushes back" the character onto the input stream *fp*; a subsequent call to *getc*, *fscanf*, etc., will encounter *c*. Only one character of push-back per file is permitted.

Low-Level I/O

This section describes the bottom level of I/O on the HP-UX system. The lowest level of I/O in HP-UX provides no buffering or any other services; it is in fact a direct entry into the operating system. You are entirely on your own, but on the other hand, you have the most control over what happens. And since the calls and usage are quite simple, this isn't as bad as it sounds.

File Descriptors

In the HP-UX operating system, all input and output is done by reading or writing files, because all peripheral devices, even the user's terminal, are files in the file system. This means that a single, homogeneous interface handles all communication between a program and peripheral devices.

In the most general case, before reading or writing a file, it is necessary to inform the system of your intent to do so, a process called "opening" the file. If you are going to write on a file, it may also be necessary to create it. The system checks your right to do so (Does the file exist? Do you have permission to access it?), and if all is well, returns a small positive integer called a **file descriptor**. Whenever I/O is to be done on the file, the file descriptor is used instead of the name to identify the file. (This is roughly analogous to the use of `READ(5, ...)` and `WRITE(6, ...)` in FORTRAN) All information about an open file is maintained by the system; the user program refers to the file only by the file descriptor.

The file pointers discussed earlier are similar in spirit to file descriptors, but file descriptors are more fundamental. A file pointer is a pointer to a structure that contains, among other things, the file descriptor for the file in question.

Since input and output involving the user's terminal are so common, special arrangements exist to make this convenient. When the command interpreter (the "shell") runs a program, it opens three files, with file descriptors 0 (*stdin*), 1 (*stdout*), and 2 (*stderr*), called the standard input, standard output, and standard error. All of these are normally connected to the terminal, so if a program reads file descriptor 0 and writes file descriptors 1 and 2, it can do terminal I/O without needing to open extra files.

If I/O is redirected to and from files with < and >, as in

```
prog <infile >outfile
```

the shell changes the default assignments for file descriptors 0 and 1 from the terminal to the named files. Similar observations hold if the input or output is associated with a pipe. Normally, file descriptor 2 remains attached to the terminal so error messages can go there. In all cases, the file assignments are changed by the shell; not by the program. The program does not need to know where its input comes from nor where its output goes, as long as it uses file 0 for input and 1 and 2 for output.

Read and Write

All input and output is done by two functions called *read* and *write*. For both, the first argument is a file descriptor. The second argument is a buffer in your program where the data is to come from or go to. The third argument is the number of bytes to be transferred. The calls are

```
n_read = read(fd, buf, n);

n_written = write(fd, buf, n);
```

Each call returns a byte count which is the number of bytes actually transferred. On reading, the number of bytes returned may be less than the number asked for, because fewer than *n* bytes remained to be read. (When the file is a terminal, *read* normally reads only up to the next newline, which is generally less than what was requested.) A return value of zero bytes implies end of file, and *-1* indicates an error of some sort. For writing, the returned value is the number of bytes actually written; it is generally an error if this isn't equal to the number supposed to be written.

The number of bytes to be read or written is quite arbitrary. The two most common values are 1, which means one character at a time ("unbuffered"), and 512, which is a convenient buffer size. Buffered 512-byte blocks are more efficient, but one-character-at-a-time I/O is not inordinately inefficient.[1]

[1] Some character special files insist on reads or writes of a specified or minimum size. Refer to the appropriate *HP-UX Reference* page for more information.

By combining these concepts, we can write a simple program to copy from a specified input file to a specified output file. This program can copy almost anything to anything by specifying redirected input and output files.

```
#define BUFSIZE 512        /* best size for HP-UX */

main()                     /* copy input to output */
{
    char buf[BUFSIZE];
    int  n;

    while ((n = read(0, buf, BUFSIZE)) > 0)
        write(1, buf, n);
    exit(0);
}
```

If the file size is not a multiple of *BUFSIZE*, some *read* will return a smaller number of bytes to be written by *write*; the next call to *read* after that will return zero.

It is instructive to see how *read* and *write* can be used to construct higher level routines like *getchar*, *putchar*, etc. For example, here is a version of *getchar* which does unbuffered input.

```
#define CMASK 0377     /* for making char's > 0 */
getchar()         /* unbuffered single character input */
{
    char c;

    return((read(0, &c, 1) > 0) ? c & CMASK : EOF);
}
```

c **must** be declared *char*, because *read* accepts a character pointer. The character being returned must be masked with *0377* to ensure that it is positive; otherwise sign extension may make it negative. (The constant *0377* is appropriate for HP computers, but not necessarily for other computers and systems.)

The second version of *getchar* does input in big chunks, and hands out the characters, one at a time.

```
#define CMASK 0377          /* for making char's > 0 */
#define BUFSIZE 512
getchar()                   /* buffered version */
{
    static char         buf[BUFSIZE];
    static char         *bufp = buf;
    static int          n = 0;

    if (n == 0) {           /* buffer is empty */
        n = read(0, buf, BUFSIZE);
        bufp = buf;
    }
    return((--n >= 0) ? *bufp++ & CMASK : EOF);
}
```

Open, Close, Unlink

Other than the default standard input, output and error files, you must explicitly open files in order to read or write them. There are two system entry points for this, *open* and *creat*.

Open is rather like the *fopen* discussed in the previous section, except that instead of returning a file pointer, it returns a file descriptor, which is just an *int*.

```
int fd;

fd = open(name, oflags);
```

As with *fopen*, the *name* argument is a character string corresponding to the external file name. The **oflags** argument is different. It consists of one or more flags that are logical ORed to indicate what types of file operations are to be allowed while the file is open. One of the three flags **O_RDONLY** (open for read only), **O_WRONLY** (open for write only), or **O_RDWR** (open for read/write) **must** be included. Refer to *open*(2) in the *HP-UX Reference* for a complete list of flags, some of which can be changed while the file is open. *open* returns *-1* if any error occurs; otherwise it returns a valid file descriptor.

If you need to *open* a file that does not exist, use a third argument to specify the file mode as follows:

```
fd = open(name, oflags, mode);
```

As before, *open* returns a file descriptor if it was able to create the file called *name*, or *-1* if not. If the file already exists, *open* truncates it to zero length. **Mode** defines the access mode that is to be assigned to the file if the file does not already exist.

In the HP-UX file system, **mode** defines nine bits of protection information associated with a file that control read, write, and execute permission for the owner of the file, for the owner's group, and for all others. Thus a three-digit octal number is most convenient for specifying the permissions. For example, 0755 specifies read, write and execute permission for the owner, and read and execute permission for the group and everyone else.

To illustrate, here is a simplified version of the HP-UX utility *cp*, a program which copies one file to another. (The main simplification is that our version copies only one file and does not permit the second argument to be a directory.)

```
#define NULL 0
#define BUFSIZE 512
#define PMODE 0644      /* RW for owner, R for group, others */

main(argc, argv)           /* cp: copy f1 to f2 */
int argc;
char *argv[];
{
    int f1, f2, n;
    char buf[BUFSIZE];

    if (argc != 3)
        error("Usage: cp from to", NULL);
    if ((f1 = open(argv[1], O_RDONLY)) == -1)
        error("cp: can't open %s", argv[1]);
    if ((f2 = creat(argv[2], O_WRONLY, PMODE)) == -1)
        error("cp: can't create %s", argv[2]);

    while ((n = read(f1, buf, BUFSIZE)) > 0)
        if (write(f2, buf, n) != n)
            error("cp: write error", NULL);
    exit(0);
}

error(s1, s2)           /* print error message and die */
char *s1, *s2;
{
    printf(s1, s2);
    printf("\n");
    exit(1);
}
```

As we said earlier, there is a limit (typically 15-25) on the number of files which a program can have open simultaneously. Accordingly, any program which intends to process many files must be prepared to re-use file descriptors. The routine *close* breaks the connection between a file descriptor and an open file, and frees the file descriptor for use with some other file. Termination of a program via *exit* or return from the main program closes all open files.

The function *unlink(*<filename>*)* removes the file <filename> from the file system.

Random Access – Lseek

File I/O is normally sequential: each *read* or *write* takes place at a position in the file right after the previous one. When necessary, however, a file can be read or written in any arbitrary order. The system call *lseek* provides a way to move around in a file without actually reading or writing:

```
lseek(fd, offset, origin);
```

forces the current position in the file whose descriptor is *fd* to move to position *offset*, which is taken relative to the location specified by *origin*. Subsequent reading or writing will begin at that position. *offset* is a *long*; *fd* and *origin* are *ints*. *origin* can be 0, 1, or 2 to specify that *offset* is to be measured from the beginning, from the current position, or from the end of the file respectively. For example, to append to a file, seek to the end before writing:

```
lseek(fd, 0L, 2);
```

To get back to the beginning ("rewind"),

```
lseek(fd, 0L, 0);
```

Notice the *0L* argument; it could also be written as *(long) 0*.

With *lseek*, it is possible to treat files more or less like large arrays, at the price of slower access. For example, the following simple function reads any number of bytes from any arbitrary place in a file.

```
get(fd, pos, buf, n)      /* read n bytes from position pos */
int fd, n;
long pos;
char *buf;
{
    lseek(fd, pos, 0); /* get to pos */
    return(read(fd, buf, n));
}
```

Error Processing

The routines discussed in this section, and, in fact, all routines that are direct entries into the system can incur errors. Usually they indicate an error by returning a value of -1. Sometimes it is nice to know what sort of error occurred, so an external variable *errno* is provided for that purpose. Refer to *errno*(2) in the *HP-UX Reference* for a detailed listing of the possible values of *errno*. *Errno* is not cleared when no error occurs, so it should not be used unless an error has occurred. Error names are preferred. Avoid using actual error numbers contained in the file */usr/include/errno.h*.

Error names can be used by a program, for example, to determine whether an attempt to open a file failed because it did not exist or because the user lacked permission to read it. Perhaps more commonly, you may want to print the reason for failure. The routine *perror* prints a message associated with the value of *errno*. More generally, *sys_errno* is an array of character strings that can be indexed by *errno* and printed by your program.

Processes

It is often easier to use a program written by someone else than to invent one's own. This section describes how to execute a program from within another.

The "System" Function

The easiest way to execute a program from another is to use the standard library routine *system*. *System* takes one argument, a command string exactly as typed at the terminal (except for the newline at the end) and executes it. For instance, to time-stamp the output of a program,

```
main()
{
    system("date");
    /* rest of processing */
}
```

If the command string has to be built from pieces, the in-memory formatting capabilities of *sprintf* may be useful.

Remember that *getc* and *putc* normally buffer their input; terminal I/O will not be properly synchronized unless this buffering is defeated. For output, use *fflush*; for input, see *setbuf* in the appendix.

Low-Level Process Creation – Execl and Execv

If you're not using the standard library, or if you need finer control over what happens, you will have to construct calls to other programs using the more primitive routines that the standard library's *system* routine is based on.

The most basic operation is to execute another program **without** *returning*, by using the routine *execl*. To print the date as the last action of a running program, use

```
execl("/bin/date", "date", NULL);
```

The first argument to *execl* is the **file name** of the command; you have to know where it is found in the file system. The second argument is conventionally the program name (that is, the last component of the file name), but this is seldom used except as a place-holder. If the command takes arguments, they are strung out after this; the end of the list is marked by a NULL argument.

The *execl* call overlays the existing program with the new one, runs that, then exits. There is no return to the original program if *exec* succeeds.

More realistically, a program might fall into two or more phases that communicate only through temporary files. Here it is natural to make the second pass simply an *execl* call from the first.

The one exception to the rule that the original program never gets control back occurs when there is an error, for example if the file can't be found or is not executable. If you don't know where *date* is located, say

```
execl("/bin/date", "date", NULL);
execl("/usr/bin/date", "date", NULL);
fprintf(stderr, "Someone stole 'date'\n");
```

A variant of *execl* called *execv* is useful when you don't know in advance how many arguments there are going to be. The call is

```
execv(filename, argp);
```

where *argp* is an array of pointers to the arguments; the last pointer in the array must be NULL so *execv* can tell where the list ends. As with *execl*, *filename* is the file in which the program is found, and *argp[0]* is the name of the program. (This arrangement is identical to the *argv* array for program arguments.)

Neither of these routines provides the niceties of normal command execution. There is no automatic search of multiple directories — you have to know precisely where the command is located. Nor do you get the expansion of metacharacters like <, >, *, ?, and *[]* in the argument list. If you want these, use *execl* to invoke the shell *sh*, which then does all the work. Construct a string *commandline* that contains the complete command as it would have been typed at the terminal, then say

```
execl("/bin/sh", "sh", "-c", commandline, NULL);
```

The shell is assumed to be at a fixed place, */bin/sh*. Its argument *−c* says to treat the next argument as a whole command line, so it does just what you want. The only problem is in constructing the right information in *commandline*.

Control of Processes – Fork and Wait

So far what we've talked about isn't really all that useful by itself. Now we will show how to regain control after running a program with *execl* or *execv*. Since these routines simply overlay the new program on the old one, to save the old one requires that it first be split into two copies; one of these can be overlaid, while the other waits for the new, overlaying program to finish. The splitting is done by a routine called *fork*:

```
proc_id = fork();
```

splits the program into two copies, both of which continue to run. The only difference between the two is the value of *proc_id*, the "process id." In one of these processes (the "child"), *proc_id* is zero. In the other (the "parent"), *proc_id* is non-zero; it is the process number of the child. Thus the basic way to call, and return from, another program is

```
if (fork() == 0)
    execl("/bin/sh", "sh", "-c", cmd, NULL);      /* in child */
```

And in fact, except for handling errors, this is sufficient. The *fork* makes two copies of the program. In the child, the value returned by *fork* is zero, so it calls *execl* which does the *command* and then dies. In the parent, *fork* returns non-zero so it skips the *execl*. (If there is any error, *fork* returns *-1*).

More often, the parent wants to wait for the child to terminate before continuing itself. This can be done with the function *wait*:

```
int status;

if (fork() == 0)
    execl(. . .);
wait(&status);
```

This still doesn't handle any abnormal conditions, such as a failure of the *execl* or *fork*, or the possibility that there might be more than one child running simultaneously. (The *wait* returns the process id of the terminated child, if you want to check it against the value returned by *fork*.) Finally, this fragment doesn't deal with any funny behavior on the part of the child (which is reported in *status*). Still, these three lines are the heart of the standard library's *system* routine, which we'll show in a moment.

The *status* returned by wait encodes in its low-order eight bits the system's idea of the child's termination status; it is 0 for normal termination and non-zero to indicate various kinds of problems. The next higher eight bits are taken from the argument of the call to *exit* which caused a normal termination of the child process. It is good coding practice for all programs to return meaningful status.

When a program is called by the shell, the three file descriptors 0, 1, and 2 (*stdin*, *stdout*, and *stderr*) are set up pointing at the right files, and all other possible file descriptors are available for use. When the program called by the shell calls another program, correct etiquette suggests making sure the same conditions hold. Open files are not affected in any way by *fork* or *exec* calls unless the close-on-exec flag has been set (see *fcntl*(2) in the *HP-UX Reference*). If the parent is buffering output that must come out before output from the child, the parent must flush its buffers before the *execl*. Conversely, if a caller buffers an input stream, the called program will lose any information that has been read by the caller.

Pipes

A **pipe** is an I/O channel intended for use between two cooperating processes: one process writes into the pipe, while the other reads. The system looks after buffering the data and synchronizing the two processes. Most pipes are created by the shell, as in

```
ls | pr
```

which connects the standard output of *ls* to the standard input of *pr*. Sometimes, however, it is most convenient for a process to set up its own plumbing; in this section, we will illustrate how the pipe connection is established and used.

The system call *pipe* creates a pipe. Since a pipe is used for both reading and writing, two file descriptors are returned; the actual usage is like this:

```
int  fd[2];

stat = pipe(fd);
if (stat == -1)
    /* there was an error . . . */
```

Fd is an array of two file descriptors, where *fd[0]* is the read side of the pipe and *fd[1]* is for writing. These can be used in *read*, *write* and *close* calls just like any other file descriptors.

If O_NDELAY is not set (see *read*(2) and *write*(2) in *HP-UX Reference*) and a process reads a pipe which is empty, the process will wait until data arrives; if a process writes into a pipe that is too full, the process will wait until the pipe empties somewhat. If the write side of the pipe is closed, a subsequent *read* will encounter end of file. If O_NDELAY is set, *read* and *write* both return immediately with the value 0.

To illustrate the use of pipes in a realistic setting, let us write a function called *popen(cmd, mode)*, which creates a process *cmd* (just as *system* does), and returns a file descriptor that will either read or write that process, according to *mode*. That is, the call

```
fout = popen("pr", WRITE);
```

creates a process that executes the *pr* command; subsequent *write* calls using the file descriptor *fout* will send their data to that process through the pipe.

Popen first creates the the pipe with a *pipe* system call; it then *fork*s to create two copies of itself. The child decides whether it is supposed to read or write, closes the other side of the pipe, then calls the shell (via *execl*) to run the desired process. The parent likewise closes the end of the pipe it does not use. These closes are necessary to make end-of-file tests work properly. For example, if a child that intends to read fails to close the write end of the pipe, it will never see the end of the pipe file, just because there is one writer potentially active.

```
#include <stdio.h>

#define READ 0
#define WRITE 1
#define tst(a, b) (mode == READ ?  (b) : (a))
static int popen_pid;

popen(cmd, mode)
char *cmd;
int  mode;
{
    int p[2];

    if (pipe(p) < 0)
        return(NULL);
    if ((popen_pid = fork()) == 0) {
        close(tst(p[WRITE], p[READ]));
        close(tst(0, 1));
        dup(tst(p[READ], p[WRITE]));
        close(tst(p[READ], p[WRITE]));
        execl("/bin/sh", "sh", "-c", cmd, 0);

        _exit(1); /* disaster has occurred if we get here */
    }
    if (popen_pid == -1)
        return(NULL);
    close(tst(p[READ], p[WRITE]));
    return(tst(p[WRITE], p[READ]));
}
```

The sequence of *closes* in the child is a bit tricky. Suppose that the task is to create a child process that will read data from the parent. Then the first *close* closes the write side of the pipe, leaving the read side open. The lines

```
close(tst(0, 1));
dup(tst(p[READ], p[WRITE]));
```

are the conventional way to associate the pipe descriptor with the standard input of the child. The *close* closes file descriptor 0, that is, the standard input. *dup* is a system call that returns a duplicate of an already open file descriptor. File descriptors are assigned in increasing order and the first available one is returned, so the effect of the *dup* is to copy the file descriptor for the pipe (read side) to file descriptor 0; thus the read side of the pipe becomes the standard input. (Yes, this is a bit tricky, but it's a standard idiom.) Finally, the old read side of the pipe is closed.

A similar sequence of operations takes place when the child process is supposed to write from the parent instead of reading. You may find it a useful exercise to step through that case.

The job is not quite done, for we still need a function *pclose* to close the pipe created by *popen*. The main reason for using a separate function rather than *close* is that it is desirable to wait for the termination of the child process. First, the return value from *pclose* indicates whether the process succeeded. Equally important when a process creates several children is that only a bounded number of unwaited-for children can exist, even if some of them have terminated; performing the *wait* lays the child to rest. Thus:

```
#include <signal.h>

pclose(fd) /* close pipe fd */
int fd;
{
    register r, (*hstat)(), (*istat)(), (*qstat)();
    int  status;
    extern int popen_pid;

    close(fd);
    istat = signal(SIGINT, SIG_IGN);
    qstat = signal(SIGQUIT, SIG_IGN);
    hstat = signal(SIGHUP, SIG_IGN);
    while ((r = wait(&status)) != popen_pid && r != -1);
    if (r == -1)
        status = -1;
    signal(SIGINT, istat);
    signal(SIGQUIT, qstat);
    signal(SIGHUP, hstat);
    return(status);
}
```

The calls to *signal* make sure that no interrupts, etc., interfere with the waiting process; this is the topic of the next section.

The routine as written has the limitation that only one pipe can be open at once, because of the single shared variable *popen_pid*; it really should be an array indexed by file descriptor. A *popen* function, with slightly different arguments and return value is available as part of the standard I/O library discussed below. As currently written, it shares the same limitation.

Signals – Interrupts and All That

This section is concerned with how to deal gracefully with signals from the outside world (like interrupts), and with program faults. Since there's nothing very useful that can be done from within C about program faults, which arise mainly from illegal memory references or from execution of peculiar instructions, we'll discuss only the outside-world signals:

Interrupt	Sent when the Interrupt character is typed (user selectable, usually DEL);
Quit	Generated by the Quit character (user selectable, usually `File Separator` character obtained by CTRL-\);
Hangup	Caused by hanging up the phone; and
Terminate	Generated by the *kill* command.

Unless other arrangements have been made (see *setpgrp*(2) and *signal*(2)), when one of these events occurs, the signal is sent to all processes that were started from the corresponding terminal, terminating the process(es). In the *quit* case, a core image file is written for debugging purposes.

The routine that alters the default action is called **signal**. It has two arguments: the first specifies the signal while the second specifies how to treat it. The first argument is just a number code; the second is an address consisting of either a function or a code requesting that the signal either be ignored or that it be given the default action. The include file *signal.h* provides names for the various arguments, and should always be included when signals are used. Thus,

```
#include <signal.h>
  ...
signal(SIGINT, SIG_IGN);
```

causes interrupts to be ignored, while

```
signal(SIGINT, SIG_DFL);
```

restores the default action of process termination. In all cases, *signal* returns the previous value of the signal catcher. The second argument to *signal* can be the name of a function (which has to be declared explicitly if the compiler hasn't seen it already). In this case, the named routine will be called when the signal occurs. Most commonly this facility is used to allow the program to clean up unfinished business before terminating, for example to delete a temporary file:

```
#include <signal.h>

main()
{
    int onintr();

    if (signal(SIGINT, SIG_IGN) != SIG_IGN)
        signal(SIGINT, onintr);

    /* Process . . .*/

    exit(0);
}

onintr( )
{
    unlink(tempfile);
    exit(1);
}
```

Why the test and the double call to *signal*? Recall that signals like interrupt are sent to **all** processes started from a particular terminal. Accordingly, when a program is to be run non-interactively (started by `&|t`), the shell turns off interrupts for it so it won't be stopped by interrupts intended for foreground processes. If this program began by announcing that all interrupts were to be sent to the *onintr* routine regardless, that would undo the shell's effort to protect it when run in the background.

The solution, shown above, is to test the state of interrupt handling, and to continue to ignore interrupts if they are already being ignored. The code as written depends on the fact that *signal* returns the previous state of a particular signal. If signals were already being ignored, the process should continue to ignore them; otherwise, they should be caught.

A more sophisticated program may wish to intercept an interrupt and interpret it as a request to stop what it is doing and return to its own command-processing loop. Think of a text editor: interrupting a long printout should not cause it to terminate and lose the work already done. The outline of the code for this case is probably best written like this:

```
#include <signal.h>
#include <setjmp.h>
jmp_buf
sjbuf;

main( )
{
   int (*istat)( ), onintr( );

   istat = signal(SIGINT, SIG_IGN);    /* save original status */
   setjmp(sjbuf);                /* save current stack position */
   if (istat != SIG_IGN)
      signal(SIGINT, onintr);

   /* main processing loop */
}

onintr( )
{
   printf("\nInterrupt\n");
   longjmp(sjbuf);          /* return to saved state */
}
```

The include file *setjmp.h* declares a type *jmp_buf* which is an object in which the state can be saved. *sjbuf* is an object of type *jmp_buf* where the *setjmp* routine saves the state of things. When an interrupt occurs, a call is forced to the *onintr* routine, which can print a message, set flags, or whatever. *Longjmp* takes as argument an object containing information placed there by *setjmp*, and restores control to the location after the call to *setjmp*, such that control (and the stack level) pop back to the place in the main routine where the signal is set up and the main loop entered. Note, incidentally, that the signal gets set again after an interrupt occurs. This is necessary because most signals are automatically reset to their default action when they occur.

Some programs that want to detect signals simply can't be stopped at an arbitrary point, such as while updating a linked list. If the routine called on occurrence of a signal sets a flag then returns instead of calling *exit* or *longjmp*, execution continues exactly where the interrupt occurred. The interrupt flag can then be tested later.

There is one difficulty associated with this approach. Suppose the program is reading the terminal when the interrupt is sent. The specified routine is duly called; it sets its flag and returns. If it were really true, as we said above, that "execution resumes at the exact point it was interrupted", the program would continue reading the terminal until the user typed another line. This behavior might well be confusing, since the user might not know that the program is reading; he presumably would prefer to have the signal take effect instantly. The method chosen to resolve this difficulty is to terminate the terminal read when execution resumes after the signal, returning an error code which indicates what happened.

Thus, programs that catch and resume execution after signals should be prepared for "errors" caused by interrupted system calls (the ones to watch out for are reads from a terminal, *wait*, and *pause*). A program whose *onintr* program only sets *intflag*, resets the interrupt signal, then returns, should usually include code like the following when it reads the standard input:

```
if (getchar( ) == EOF)
   if (intflag)
                /* EOF caused by interrupt */
   else
                /* true end-of-file */
```

Another aspect of error handling that must be dealt with is associated with programs where the user has elected to catch an asynchronous signal such as an interrupt or quit signal, and the signal occurs during a system call, producing the error EINTR. If execution is resumed after processing the signal, it will appear as if the interrupted system call returned the EINTR error unless the system call is restarted. Refer to *sigvector*(2) in the *HP-UX Reference* for more information.

A final subtlety to keep in mind becomes important when signal-catching is combined with execution of other programs. Suppose a program catches interrupts, and also includes a method (like "!" in the editor) whereby other programs can be executed. Then the code should look something like this:

```
if (fork( ) == 0)
     execl(. . . );
signal(SIGINT, SIG_IGN);    /* ignore interrupts */
wait(&status);              /* until the child is done */
signal(SIGINT, onintr);     /* restore interrupts */
```

Why is this? Again, it's not obvious but not really difficult. Suppose the program you call catches its own interrupts. If you interrupt the subprogram, it will get the signal and return to its main loop, and probably read your terminal. But the calling program will also pop out of its wait for the subprogram and read your terminal. Having two processes reading your terminal is very unfortunate, since the system figuratively flips a coin to decide who should get each line of input. A simple way out is to have the parent program ignore interrupts until the child is done. This reasoning is reflected in the standard I/O library function *system*:

```
#include <signal.h>

system(s)    /* run command string s */
char *s;
{
     int status, pid, w;
     register int (*istat)( ), (*qstat)( );

     if ((pid = fork( )) == 0) {
          execl("/bin/sh", "sh", "-c", s, 0);
          _exit(127);
     }
     istat = signal(SIGINT, SIG_IGN);
     qstat = signal(SIGQUIT, SIG_IGN);
     while ((w = wait(&status)) != pid && w != -1)
          ;
     if (w == -1)
          status = -1;
     signal(SIGINT, istat);
     signal(SIGQUIT, qstat);
     return(status);
}
```

As an aside on declarations, the function *signal* obviously has a rather strange second argument. It is in fact a pointer to a function delivering an integer, and this is also the type of the signal routine itself. The two values *SIG_IGN* and *SIG_DFL* have the right type, but are chosen so they coincide with no possible actual functions.

Appendix – The Standard I/O Library

The standard I/O library was designed with the following goals in mind.

- It must be as efficient as possible, both in time and in space, so that there will be no hesitation in using it no matter how critical the application.
- It must be simple to use, and also free of the magic numbers and mysterious calls whose use mars the understandability and portability of many programs using older packages.
- The interface provided should be applicable on all machines, whether or not the programs which implement it are directly portable to other systems, or to machines other than the one upon which the program was written.

General Usage

Each program using the library must have the line

```
#include <stdio.h>
```

which defines certain macros and variables. The routines are in the normal C library, so no special library argument is needed for loading. All names in the include file intended only for internal use begin with an underscore (_) to reduce the possibility of collision with a user name. The names intended to be visible outside the package are

stdin	The name of the standard input file
stdout	The name of the standard output file
stderr	The name of the standard error file
EOF	is actually -1, and is the value returned by the read routines on end-of-file or error.
NULL	is a notation for the null pointer, returned by pointer-valued functions to indicate an error
FILE	expands to *struct _iob* and is a useful shorthand when declaring pointers to streams.

BUFSIZ is a number (viz. 512) of the size suitable for an I/O buffer supplied by the user. See *setbuf*, below.

getc, *getchar*, *putc*, *putchar*, *feof*, *ferror*, *fileno* are defined as macros. Their actions are described below; they are mentioned here to point out that it is not possible to redeclare them and that they are not actually functions; thus, for example, they cannot have breakpoints set on them. Also, watch out for side-effects if an expression is used as an argument because it might get evaluated more than once, producing rather bizarre (and very incorrect) results.

The routines in this package offer the convenience of automatic buffer allocation and output flushing where appropriate. The names *stdin*, *stdout*, and *stderr* are, in effect, constants and cannot be assigned to.

Calls

`FILE *fopen(<filename>, <type>) char *<filename>, *<type>;`
opens the file and, if needed, allocates a buffer for it. <filename> is a character string specifying the name. <type> is a character string (not a single character). It can be "**r**", "**w**", or "**a**" to indicate intent to read, write, or append. The value returned is a file pointer. If it is NULL, the attempt to open failed.

`FILE *freopen(filename, type, ioptr) char *filename, *type; FILE *ioptr;`
closes the stream named by *ioptr*, if necessary, then reopens it as if by *fopen*. If the attempt to open fails, NULL is returned. Otherwise *ioptr*, now refers to the new file. Often the reopened stream is *stdin* or *stdout*.

`FILE *fdopen (fildes, type) int fildes; char *type;`
associates the stream named by *ioptr* with a file descriptor obtained from *open*, *dup*, *creat*, or *pipe*(2) which open files but do not return pointers to a stream FILE structure *ioptr*. Streams are required input for several library routines described in Section 3 of the *HP-UX Reference*.

`int getc(ioptr) FILE *ioptr;`
returns the next character from the stream named by <ioptr>, which is a pointer to a file such as returned by *fopen*, or the name *stdin*. The integer *EOF* is returned on end-of-file or when an error occurs. The null character is a legal character.

`int fgetc(ioptr) FILE *ioptr;`
acts like *getc* but is a genuine function, not a macro, so it can be pointed to, passed as an argument, etc.

`putc(c, ioptr) FILE *ioptr;`
writes the character c on the output stream named by *ioptr*, which is a value returned from *fopen* or perhaps *stdout* or *stderr*. The character is returned as value, but EOF is returned on error.

`fputc(c, ioptr) FILE *ioptr;`
acts like *putc* but is a genuine function, not a macro.

`fclose(ioptr) FILE *ioptr;`
closes the file corresponding to *ioptr* after any buffers are emptied. Any buffering allocated by the I/O system is freed. *fclose* is automatic on normal termination of the program.

`fflush(ioptr) FILE *ioptr;`
writes out any buffered information on the (output) stream named by *ioptr*. Output files are normally buffered if and only if they are not directed to the terminal; however, *stderr* always starts off unbuffered and remains so unless *setbuf* is used, or unless it is reopened.

`exit(errcode);`
terminates the process and returns its argument as status to the parent. This is a special version of the routine which calls *fflush* for each output file. To terminate without flushing, use *_exit*.

`feof(ioptr) FILE *ioptr;`
returns non-zero when end-of-file has occurred on the specified input stream.

`ferror(ioptr) FILE *ioptr;`
returns non-zero when an error has occurred while reading or writing the named stream. The error indication lasts until the file has been closed.

`getchar( );`
is identical to `getc(stdin)`.

`putchar(c);`
is identical to `putc(c, stdout)`.

`char *fgets(s, n, ioptr) char *s; FILE *ioptr;`
reads up to $n-1$ characters from the stream *ioptr* into the character pointer *s*. The read terminates with a newline character. The newline character is placed in the buffer followed by a null character. *Fgets* returns the first argument, or NULL if error or end-of-file occurred.

`fputs(s, ioptr) char *s; FILE *ioptr;`
writes the null-terminated string (character array) *s* on the stream *ioptr*. No newline is appended. No value is returned.

`ungetc(c, ioptr) FILE *ioptr;`
pushes the argument character *c* back on the input stream named by *ioptr*. Only one character can be pushed back.

`printf(format, a1, . . . ) char *format;`
`fprintf(ioptr, format, a1, . . . ) FILE *ioptr; char *format;`
`sprintf(s, format, a1, . . . )char *s, *format;`
printf writes on the standard output. *fprintf* writes on the named output stream. *sprintf* puts characters in the character array (string) named by *s*. The specifications are as described in section *printf* (3) of the *HP-UX Reference*.

`scanf(format, a1, . . . ) char *format;`
`fscanf(ioptr,\ format,\ a1, . . . ) FILE *ioptr; char *format;`
`sscanf(s, format, a1, . . . ) char *s, *format;`
scanf reads from the standard input. *fscanf* reads from the named input stream. *sscanf* reads from the character string supplied as *s*. *Scanf* reads characters, interprets them according to a format, and stores the results in its arguments. Each routine expects as arguments a control string *format*, and a set of arguments, **each of which must be a pointer,** indicating where the converted input should be stored.

Scanf returns as its value the number of successfully matched and assigned input items. This can be used to decide how many input items were found. On end of file, EOF is returned; note that this is different from 0, which means that the next input character does not match what was called for in the control string.

`fread(ptr, sizeof(*ptr), nitems, ioptr) FILE *ioptr;`
reads *nitems* of data beginning at *ptr* from file *ioptr*. No advance notification that binary I/O is being done is required; when, for portability reasons, it becomes required, it will be done by adding an additional character to the mode-string on the *fopen* call.

`fwrite(ptr, sizeof(*ptr), nitems, ioptr) FILE *ioptr;`
like *fread*, but in the other direction.

`rewind(ioptr) FILE *ioptr;`
rewinds the stream named by *ioptr*. It is not very useful except on input, since a rewound output file is still open only for output.

`system(string) char *string;`
string is executed by the shell as if typed at the terminal.

`getw(ioptr) FILE *ioptr;`
returns the next 32-bit word from the input stream named by *ioptr*. EOF is returned on end-of-file or error, but since this a perfectly good integer *feof* and *ferror* should be used.

`putw(w, ioptr) FILE *ioptr;`
writes the integer *w* on the named output stream.

`setbuf(ioptr, buf) FILE *ioptr; char *buf;`
setbuf can be used after a stream has been opened but before I/O has started. If *buf* is NULL, the stream will be unbuffered. Otherwise the buffer supplied will be used. It must be a character array of sufficient size: `char buf[BUFSIZ];`

`int setvbuf(ioptr, buf, type, size) FILE *ioptr; char *buf; int type, size;`
setvbuf can be used after a stream has been opened but before I/O has started. *Type* defines the type of buffer to be used: fully buffered, line buffered, or completely unbuffered; while *size* defines the buffer size. See *setbuf*(3S) in *HP-UX Reference* for more information.

`fileno(ioptr) FILE *ioptr;`
returns the integer file descriptor associated with the file.

`fseek(ioptr, offset, ptrname) FILE *ioptr; long offset;`
adjusts the location of the next byte in the stream named by *ioptr*. *offset* is a long integer. If *ptrname* is 0, the offset is measured from the beginning of the file; if *ptrname* is 1, the offset is measured from the current read or write pointer; if *ptrname* is 2, the offset is measured from the end of the file. The routine accounts properly for any buffering. (When this routine is used on HP-UX systems, the offset must be a value returned from *ftell* and the ptrname must be 0).

`long ftell(ioptr) FILE *ioptr;`
returns the byte offset (measured from the beginning of the file) associated with the named stream. Any buffering is properly accounted for. (On HP-UX systems the value of this call is useful only for handing to *fseek*, so as to position the file to the same place it was when *ftell* was called.)

`getpw(uid, buf) char *buf;`
searches the password file for the given integer user ID. If an appropriate line is found, it is copied into the character array *buf*, and 0 is returned. If no line is found corresponding to the user ID then 1 is returned.

`char *malloc(num);`
allocates *num* bytes. The pointer returned is sufficiently well aligned to be usable for any purpose. NULL is returned if no space is available.

`char *calloc(num, size);`
allocates space for *num* items each of size *size*. The space is guaranteed to be set to 0 and the pointer is sufficiently well aligned to be usable for any purpose. NULL is returned if no space is available .

`cfree(ptr) char *ptr;`
Space is returned to the pool used by *calloc*. Disorder can be expected if the pointer was not obtained from `calloc`.

The following are macros whose definitions can be obtained by including <*ctype.h*>.

`isalpha(c)` returns non-zero if the argument is alphabetic.

`isupper(c)` returns non-zero if the argument is uppercase alphabetic.

`islower(c)` returns non-zero if the argument is lowercase alphabetic.

`isdigit(c)` returns non-zero if the argument is a digit.

`isspace(c)` returns non-zero if the argument is a spacing character such as tab, space (blank), newline, vertical tab, form-feed, or other white-space character.

`ispunct(c)` returns non-zero if the argument is any punctuation character, i.e., not a space, letter, digit or control character.

`isalnum(c)` returns non-zero if the argument is a letter or a digit.

`isprint(c)` returns non-zero if the argument is printable-a letter, digit, or punctuation character.

`iscntrl(c)` returns non-zero if the argument is a control character.

`isascii(c)` returns non-zero if the argument is an ASCII character, i.e., less than octal 0200.

`toupper(c)` returns the uppercase character corresponding to the lowercase letter *c*.

`tolower(c)` returns the lowercase character corresponding to the uppercase letter

Notes

Index

a

access, file 5
access mode, file 14
argc 2
argument count 2
arguments 2
argv 2

b

buffer flushing 8

c

cat 4
catching interrupts 27
child process 18
close 15
close sequences in child 22
closed pipe 20, 21
creat 13
ctype.h 33

d

dup 22

e

empty pipe 20
end-of-file returns 6
errno 16
errno.h 16
error names 16
error processing 16
exec 20
execl 17, 18, 21
execute without returning 17

execv 18
existing file truncated by open 14
exit 4, 8, 18, 25
exit calls fclose 8
_exit to terminate parent and child 8

f

failure of fork or execl 18
fclose 8
fflush 9, 17
fgets 9
file access 5
file access mode 14
file access, random 15
file descriptor 10
file pointer 5, 10
file rewind 15
file treated as array 15
files open simultaneously 15
fopen 5, 6, 13
fork 18, 20, 21
fp 5, 9
fprintf 7
fputs 9
fscanf 9

g

getc 6, 9, 17
getchar 3

h

hangup signal 23

i

interrupt catching 27
interrupt signal 23
interrupts 23
intflg 26
I/O calls 29

I/O library 5, 28
I/O macros in ctype.h 33
I/O redirection 11

l

library, standard I/O 5
longjmp 25
low-level I/O 10
lseek 15

m

main 2
metacharacter expansion not available 18

o

offset, used in random file access 15
onintr routine 24, 26
open 13
open files, simultaneous 15
opening files 5
O_RDONLY flag 13
O_RDWR flag 13
origin for random file access 15
O_WRONLY flag 13

p

parent process 18
pause 26
pclose, close pipe 22
perror, print error 16
pipes 20
popen, open pipe 21, 23
popen_pid 23
printf 3
process control 18
proc_id 18
program arguments 2
push back characters 9
putc 6, 8, 17
putchar 3

q

quit signal 23

r

random file access 15
re-use of file descriptors 15
read 11, 20
redirection (file substitution) 3
rewind file 15

s

scanf 3
setbuf 17
setjmp.h 25
setpgrp(2) 23
signal 23
signal command arguments 27
signal(2) 23
signal.h 23
signals 23–27
sigvector(2) 26
simultaneous open files 15
sprintf 9, 17
sscanf 9
standard error 10, 20
standard input 3, 10, 20
standard I/O library 5
standard output 3, 10, 20
status returned by wait 18
stderr 6, 8, 20
stderr file descriptor 10, 20
stdin 6, 20
stdin file descriptor 10, 20
stdio.h 5
stdout 6, 20
stdout file descriptor 10, 20
sub-process, exit as a 8
sys_errno, error name string array 16
system function 17

t

terminal I/O 10
terminate signal 23

u

ungetc 9
unlink 15

w

wait 18, 22, 26
wait for child process 18
wc 5
word count 5
write 11, 20

Table of Contents

Using C Library Routines

Part 1: Standard Input/Output Routines

Input/Output Using Stdin and Stdout 5
Single-character Input/Output 5
String Input/Output 6
Formatted Input/Output 7
Input/Output from/to Strings 21
Reading Data from a String 21
Writing Data Into a String 24
Input/Output Using Ordinary Files 26
Opening Ordinary Files 26
Single-Character Input/Output 29
Character Push-Back 32
String Input/Output 33
Formatted Input/Output 35
Binary Input/Output 36
Stream Status and Control Routines 42
Stream Status Inquiry Routines 42
Re-positioning Stream I/O Operations 45
Stream Control Routines 50
Converting Between File Pointers and File Descriptors 55
Inter-Process Communication 58

Part 2: Math Routines

Absolute Value Functions 62
Power, Square Root, and Logarithmic Functions 63
Trigonometric Functions 64
Miscellaneous Functions 68
Calculating Upper and Lower Bounds 68
Calculating Remainders 69
Calculating A Hypotenuse 70
Generating Random Numbers 71
Floating-Point Exponentiation Routines 72

Part 3: String Manipulations

Character Conversion and Classification 73
Converting Between Uppercase and Lowercase 73
Character Classification 74

String Manipulation 75
Concatenating Strings 75
Copying Strings 75
Comparing Strings 77
Finding the Length of a String 79
Finding Characters in Strings 79
Miscellaneous String Routines 81
Finding Characters Common to Two Strings 81
Breaking a String into Tokens 82

Part 4: Date and Time Manipulation 83

Using C Library Routines

The purpose of this tutorial is to illustrate the use of the most commonly used library routines described in Section 3 of the *HP-UX Reference* manual. Examples are included to demonstrate programming techniques.

This article assumes that you have a working knowledge of the C programming language. No attempt is made here to explain or teach C programming techniques, other than those that are relevant to a particular library routine.

Material is presented in three sections, each dealing with the following topics in the order listed:

- Standard Input/Output Routines,
- Math Routines, including trigonometric and other functions, and
- String Manipulation Routines.

Part 1:
Standard Input/Output Routines

There are more library routines in this category than in any other. Described under this heading are routines that perform all kinds of input and output, from single characters to entire strings. Also described are routines that adjust I/O buffering, routines that enable input from or output to files, and routines that enable random access to data. These routines require that the include file **stdio.h** be **#include**d in C programs containing calls to them.

The standard I/O routines are inseparably linked with files. A file must be *opened* before its contents can be used. Three "files" are automatically opened for you by the system. Including **stdio.h** in your program assigns buffering to them. These three "files" are the *standard input*, *standard output*, and *standard error* files. Their names are **stdin**, **stdout**, and **stderr**, respectively.

Actually, it is more accurate to think of these "files" as *pipes* connecting two points. Each pipe accepts data at one end, and transfers the data to its destination at the other end. These pipes have only limited ability to store data. Once a certain number of bytes have been written into the pipe, data must be read from the other end before the pipe can accept more data. Writing data into a pipe is analogous to pumping water into a pipeline. The pipeline is able to hold some water, but if the valve at the receiving end of the pipe is shut, the pipeline is soon unable to hold any more water. Opening the valve is analogous to reading data from the pipe. Once water has been removed from the pipeline, more water can be pumped in at the source.

Once a certain volume of water has been allowed to flow out of a pipeline, that same water no longer exists in the pipeline. This is also true for data that has been received from **stdin**, **stdout**, and **stderr**. Reading data from **stdin**, for instance, removes that data from **stdin**. You can see that **stdin**, **stdout**, and **stderr** are very different from ordinary files. Not only can they store small amounts of data, but that data exists only until it is read (unless it is "pushed back" —— see *Character Push-Back* later in this article).

Stdin is opened for reading. This means that your program can only receive data from **stdin**; it cannot write data into it. By default, **stdin**'s source of data is your terminal's keyboard. Thus, whatever you type at your keyboard provides the data that flows through **stdin** and becomes available to your program at the other end. By default, **stdin** is buffered via a buffer containing exactly **BUFSIZ** bytes, where BUFSIZ is a constant defined in **stdio.h**. For Series 200 and Series 500 computers, BUFSIZ is 1024. Due to terminal driver characteristics, data you type in at your keyboard is not available to a program until you press RETURN (or its equivalent).

Stdout is opened for writing, which means that your program is the source of data for **stdout**. Your program cannot, however, read data from **stdout**. By default, the destination of **stdout** is your terminal's screen. Thus, data fed into **stdout** appears on your screen. **Stdout** is typically used for all output that arises from successful execution of a program (status reports, lists of tasks being performed, etc.). Like **stdin**, **stdout** is buffered via a buffer containing BUFSIZ bytes.

Stderr is also opened for writing, allowing your program to feed data into it, but disallowing reading. Just like **stdout**, **stderr**'s destination is your terminal's screen by default. **Stderr** is typically used to output data which arises from an erroneous condition in a program, such as error messages, warnings, etc. **Stderr** is unbuffered by default, which means that data written to **stderr** is transferred to its destination one byte at a time.

The buffering for these pipes, as well as for any open file, can be modified — see the *Stream Status and Control Routines* section later in this tutorial.

Of course, your program would be severely limited in its I/O capabilities if it had only these three pipes to work with. Therefore, ordinary text files can be opened for reading, or created/opened for writing, appending, or both reading and writing. Directories can also be opened, but only for reading. These features are discussed later in this article. For now, the use of **stdin** and **stdout** is described (**stderr** is also left for later discussion).

Input/Output Using Stdin and Stdout

This section describes those routines which are capable of I/O using **stdin** and **stdout** only. The routines discussed are *getchar* and *putchar* (single character I/O), *gets* and *puts* (string I/O), and *scanf* and *printf* (formatted I/O of all types).

Single-character Input/Output

This section describes the two basic input and output routines, *getchar* and *putchar*. *Getchar* is a macro defined in **stdio.h** which reads one character from **stdin**. Similarly, *putchar* is also a macro defined in **stdio.h**. *Putchar* writes one character on **stdout**.

As an example, consider the following program, which simply reads **stdin** and echos whatever it finds to **stdout**. The program terminates when it receives an at-sign (@) from **stdin**.

```
#include <stdio.h>
main()
{
   int c;

   while((c = getchar()) != '@')
        putchar(c);
   putchar('\n');
}
```

Why is *c* declared an **int** instead of a **char**? For most applications, **char** works fine. In certain cases, however, sign extension, bit shifting, and similar operations cause strange results with **chars**. Therefore, **int** is used here, and in all following examples, to be safe.

The final *putchar* statement in the program is used to output a new-line so that your shell prompt appears at the beginning of a new line, instead of at the end of the last line of output. Type it in and give it a try! Remember that your input is not available to the program until you press RETURN.

Getchar and *putchar* are most useful in *filters* — programs that accept data and modify it in some way before passing it on. Suppose you want to write a program that puts parentheses around each vowel encountered in the input. It's easy to do with these routines:

```
#include <stdio.h>
main()
{
   int c;

   while((c = getchar()) != '\n') {
      if(vowel(c)) {
         putchar('(');
         putchar(c);
         putchar(')');

      }else
          putchar(c);
   }
vowel(c)
char c;
{
   if(c=='a' || c=='A' || c=='e' || c=='E' || c=='i' || c=='I'
   || c=='o' || c=='O' || c=='u' || c=='U')
      return(1);
   else
      return(0);
}
```

The vowel test is placed in the function **vowel**, since it tends to clutter up the main program. This program terminates when it encounters a new-line.

String Input/Output

The *gets* function reads a string from **stdin** and stores it in a character array. The string is terminated by a new-line in the input, which *gets* replaces with a NULL character in the array. Its companion function, *puts*, copies a string from a character array to **stdout**. The string is terminated by a NULL character in the array, which *puts* replaces with a new-line in the output.

The simple "echo" program from the last section can be rewritten using *gets* and *puts*.

```
#include <stdio.h>
main()
{
   char line[80], *gets();

   while((gets(line)) != NULL)
      puts(line);
}
```

This program, as written, runs forever. To terminate it, press BREAK (or its equivalent). Later, when string comparison and string length routines are introduced, an intelligent termination condition can be written for this program.

Formatted Input/Output

The *scanf* and *printf* routines are powerful tools enabling you to read and write data in formatted form, respectively.

Scanf

Scanf is the formatted-input library routine. Its syntax is:

```
scanf(format, [item[, item ...]]);
```

where *format* is a character pointer to a character string (or the character string itself enclosed in double quotes), and *item* is the *address* of a variable.

The purpose of the format is to specify how the data to be read is presented on **stdin**, and what types of data are found there. The format consists of two things: *conversion specifications*, and literal characters.

Conversion Specifications

A conversion specification is a character sequence which tells *scanf* how to interpret the data received at that point in the input. For example, if a conversion specification says "treat the next piece of data as a decimal integer", then that data is interpreted and stored as a decimal integer.

In the format, a conversion specification is introduced by a percent sign (%), optionally followed by an asterisk (*) (called the *assignment suppression character*), optionally followed by an integer value (called the *field width*). The conversion specification is terminated by a character specifying the type of data to expect. These terminating characters are called *conversion characters*.

When a conversion specification is encountered in a format, it is matched up with the corresponding item in the item list. The data formatted by that specification is then stored in the location pointed to by that item. For example, if there are four conversion specifications in a format, the first specification is matched up with the first item, the second specification with the second item, and so on.

The number of conversion specifications in the format is directly related to the number of items specified in the item list. With one exception, there must be at least as many items as there are conversion specifications in the format. If there are too few items in the item list, an error occurs; if there are too many, the excess items are simply ignored. The one exception occurs when the assignment suppression character (*) is used. If an asterisk occurs immediately after the percent sign (before the field width, if any), then the data formatted by that conversion specification is discarded. No corresponding item is expected in the item list. This is useful for skipping over unwanted data in the input.

Conversion Characters

There are eight conversion characters available. Three of them are used to format integer data, three are used to format character data, and two are used for floating-point data.

The integer conversion characters are:

d a decimal integer is expected;

o an octal integer is expected;

x a hexadecimal integer is expected;

The character conversion characters are:

c a single character is expected;

s a character string is expected;

[a character string is expected;

The floating-point conversion characters are:

e, f a floating-point number is expected;

Integer Conversion Characters

The **d**, **o**, and **x** conversion characters read characters from **stdin** until an inappropriate character is encountered, or until the number of characters specified by the field width, if given, is exhausted (whichever comes first).

For **d**, an inappropriate character is any character *except* +, −, and 0 thru 9. For **o**, an inappropriate character is any character *except* +, −, and 0 thru 9. That's right − 8 and 9 are allowed in octal numbers! If you enter, say, 1294 to be interpreted by the **o** conversion character, it still interprets the entire number as octal, and converts the digits to the octal digit range. Thus, 1294 actually gets stored as 1314 (octal). For **x**, an inappropriate character is any character *except* +, −, 0 thru 9, and the characters a − f and A thru F. Note that negative octal and hexadecimal values are stored in their 2's complement form with sign extension. Thus, they may look unfamiliar if you print them out later (using *printf* − see below).

These integer conversion characters can be capitalized or preceded by a lower-case L (**l**) to indicate that a **long int** should be expected rather than an **int**. They can also be preceded by **h** to indicate a **short int**. The corresponding items in the item list for these conversion characters must be pointers to integer variables of the appropriate length.

Character Conversion Characters

The **c** conversion character reads the next character from **stdin**, no matter what that character is. The corresponding item in the item list must be a pointer to a character variable. If a field width is specified, then the number of characters indicated by the field width are read. In this case, the corresponding item must refer to a character array large enough to hold the characters read.

Note that strings read using the **c** conversion character are **not** automatically terminated with a NULL character in the array. Since all C library routines which utilize strings assume the existence of a NULL terminator, be sure you add the NULL character yourself. Otherwise, library routines are not able to tell where the string ends, and you'll get puzzling results.

The **s** conversion character reads a character string from **stdin** which is delimited by one or more space characters (blanks, tabs, or new-lines). If no field width is given, the input string consists of all characters from the first non-space character up to (but not including) the first space character. Any initial space characters are skipped over. If a field width is given, then characters are read, beginning with the first non-space character, up to the first space character, or until the number of characters specified by the field width is reached (whichever comes first). The corresponding item in the item list must refer to a character array large enough to hold the characters read, plus a terminating NULL character which is added automatically.

An important point to remember about the **s** conversion character is that it *cannot* be made to read a space character as part of a string. Space characters are always skipped over at the beginning of a string, and they terminate reading whenever they occur in the string. For example, suppose you want to read the first character from the following input line:

```
"          Hello, there!"
```

(10 spaces followed by "Hello, there!", the double quotes being added for clarity). If you use **%c**, you get a space character. However, if you use **%1s**, you get "H" (the first non-space character in the input).

The [conversion character also reads a character string from **stdin**. However, this character should be used when a string is *not* to be delimited by space characters. The left bracket is followed by a list of characters, and is terminated by a right bracket. If the first character after the left bracket is a circumflex (^), then characters are read from **stdin** until a character is read which matches one of the characters between the brackets. If the first character is *not* a circumflex, then characters are read from **stdin** until a character *not* occurring between the brackets is found. The corresponding item in the item list must refer to a character array large enough to hold the characters read, plus a terminating NULL character which is added automatically.

The three string conversion characters provide you with a complete set of string-reading capabilities. The **c** conversion character can be used to read *any* single character, or to read a character string *when the exact number of characters in the string is known beforehand.* The **s** conversion character enables you to read any character string *which is delimited by space characters, and is of unknown length.* Finally, the [conversion character enables you to read character strings *that are delimited by characters other than space characters, and which are of unknown length.*

Floating-Point Conversion Characters

The **e** and **f** conversion characters read characters from **stdin** until an inappropriate character is encountered, or until the number of characters specified by the field width, if given, is exhausted (whichever comes first).

Both **e** and **f** expect data in the following form: an optionally signed string of digits (possibly containing a decimal point), followed by an optional exponent field consisting of an **E** or **e** followed by an optionally signed integer. Thus, an inappropriate character is any character *except* +, −, ., 0 thru 9, E, or e.

These floating-point conversion characters may be capitalized, or preceded by a lower-case L (l), to indicate that a **double** value is expected rather than a **float**. The corresponding items in the item list for these conversion characters must be pointers to floating-point variables of the appropriate length.

Literal Characters

Any characters included in the format which are *not* part of a conversion specification are *literal characters*. A literal character is expected to occur in the input at exactly that point. Note that since the percent sign is used to introduce a conversion specification, you must type two percent signs (%%) to get a literal percent sign.

Examples. Suppose that you have to read the following line of data:

```
NAME: Joe Kool; AGE: 27; PROF: Elec Engr; SAL: 39550
```

To get the vital data, you must read two strings (containing spaces), and two integers. You also have data that should be ignored, such as the semicolons and the identifying strings ("NAME:"). How do you go about reading this?

First, note that the identifying strings are always delimited by space characters. This suggests use of the **s** conversion character to read them. Second, you can never know the exact sizes of the NAME and PROF fields, but note that they are both terminated by a semicolon. Thus, you can use [to read them. Finally, the **d** conversion character can be used to read both integers. (Note: on 16-bit processors, you probably need to use a **long int** to read the salaries. Thus, **D** or **ld** should be used instead of **d**.)

The following code fragment successfully reads this data:

```
char name[40], prof[40];
int age, salary;
   ...
scanf("%*s%*[ ]%[^;]%*c%*s%d%*c%*s%*[ ]%[^;]%*c%*s%d",name,&age,\
prof,&salary);
```

For easier understanding, break the format into pieces:

%*s	This reads the string "NAME:". Since an asterisk is given, the string is simply read and discarded.
%*[]	This gets rid of all blanks occurring between "NAME:" and the employee's name. Note that this gets rid of one or more blanks, giving the format some flexibility.
%[^;]	This reads all characters from the current character up to a semicolon, and assigns the characters to the array *name*.
%*c	This gets rid of the semicolon left over after reading the name.
%*s	This reads the next identifying string, "AGE:", and discards it.

%d This reads the integer age given, and assigns it to *age*. The semicolon after the age terminates %d, because that character is not appropriate for an integer value. Note that the address of *age* is given in the item list (&age) instead of the variable name itself. If this is not done, a memory fault occurs at run-time.

%*c This gets rid of the semicolon following the age.

%*s This reads the next identifying string, "PROF:", and discards it.

%*[] This removes all blanks between "PROF:" and the next string.

%[^;] This reads all characters up to the next semicolon, and assigns them to the character array *prof*.

%*c This gets rid of the semicolon following the profession string.

%*s This reads the final identifying string, "SAL:", and discards it.

%d This reads the final integer and assigns it to the integer variable *salary*. Again, note that the address of *salary* is given, not the variable name itself.

Although somewhat confusing to read, this format is quite flexible, since it allows for multiple spaces between items and varying identifying strings (i.e. "PROFESSION:" could be specified instead of "PROF:"). The following *scanf* call reads the same data, but is much less flexible:

```
scanf("NAME: %[^;]; AGE:%d; PROF: %[^;]; SAL: %d",name,&age,prof,&salary);
```

Here, literal characters are used to exactly match the characters in the input line. This works fine if you can be sure that the data always appears in this form. If one typing variation is made, however, such as typing "SALARY:" instead of "SAL:", the *scanf* fails.

Scanf waits for more data as long as there are unsatisfied conversion specifications in the format. Thus, a *scanf* call like

```
scanf("%f%f%f", &float1, &float2, &float3);
```

where *float1*, *float2*, and *float3* are all variables of type **float**, allows you to enter data in several ways. For example,

```
14.77 29.8 13.0
```

is read correctly by *scanf*, as is

```
14.77 [RETURN]

29.8 [RETURN]

13.0 [RETURN]
```

Note: using decimal points in floating-point data is recommended whenever floating-point variables are being read. However, *scanf* converts integer data to floating-point if the conversion specification so demands. Thus, "13.0" in the previous example could have been entered as "13" with no side effects.

As a final example, consider the input string

```
abcdef137 d14.77ghijklmnop
```

Suppose that the following code fragment is used to read this string:

```
char arr1[10], arr2[10], arr3[10], arr4[10];
float float1;
scanf("%4c%[^3]%6c%f%[ghijkl]",arr1,arr2,arr3,&float1,arr4);
```

What values are stored in the variables listed? (Give this some thought before reading on.) As before, break up the format into separate conversion specifications, and see what data is demanded by each.

%4c	reads four characters, and assigns them to *arr1*. Thus, the string "abcd" is assigned to *arr1*. Note that an extra character, NULL, is appended to the end of the string.
%[^3]	reads all characters from the current character up to the character "3". This assigns "ef1", along with an added NULL character, to the array *arr2*.
%6c	reads the next six characters and stores them in the array *arr3*. Thus, "37 d14" is assigned to *arr3*, terminated by a NULL character.
%f	reads a floating-point value which, due to the lack of a field width, is terminated by the first "inappropriate" character. Thus, the value ".77" is assigned to *float1*.
%[ghijkl]	reads all characters up to the first character not occurring between the brackets. This stores the string "ghijkl", along with an appended NULL character, in the array *arr4*.

Note that there are some characters left in **stdin** that were not read. What happens to these characters? Do they just go away? No! Any characters left unread in the input remain there! This can cause unexpected errors. Suppose that, later in the above program fragment, you want to read a string from **stdin** using **%s**. No matter what string you type in as input, it will never be read, because the **%s** conversion specification is satisfied by reading "mnop" — the characters left over from the previous read operation! To solve this, always be sure you have read the entire current line of input before attempting to read the next. To fix this in the previous *scanf* example, just add a **%*s** conversion specification at the end of the format. This reads and discards the left-over characters.

Printf

Printf is the other half of the formatted I/O team. It enables you to output data in formatted form. Its syntax is identical to that of *scanf*:

```
printf(format, [item[, item ...]\|]);
```

where the *format* is a pointer to a character string (or the character string itself enclosed in double quotes) which specifies the format and content of the data to be printed. Each *item* is a variable or expression specifying the data to print.

Printf's format is similar in many respects to that of *scanf*. It is made up of conversion specifications and literal characters. As in *scanf*, literal characters are all characters that are not part of a conversion specification. Literal characters are printed on **stdout** exactly as they appear in the format.

Literal Characters

Included in the list of literal characters are *escape sequences*, which are sequences beginning with a backslash (\e) which stand for other characters. The following list shows the escape sequences defined for *printf* (and *scanf*, though less frequently used):

\b	backspace;
\n	new-line (carriage-return/line-feed sequence); output begins at the beginning of a new line;
\r	carriage-return without a line-feed; output begins at the beginning of the current line (data already printed on that line is over-printed);
\t	tab;
\\	literal backslash;
nnn	the character represented by the octal number *nnn* in the ASCII character set. *Nnn* must begin with a zero. For example, **\007** is an ASCII bell, which beeps the bell on your terminal.

Conversion Specifications

A conversion specification for *printf* is very similar to that of *scanf*, but is a bit more complicated. The following list shows the different components of a conversion specification in their correct sequence:

1. A percent sign (%), which signals the beginning of a conversion specification; to output a literal percent sign, you must type two percent signs (%%);
2. Zero or more *flags*, which affect the way a value is printed (see below);
3. an optional decimal digit string which specifies a minimum *field width*;
4. an optional *precision* consisting of a dot (.) followed by a decimal digit string;
5. an optional **l** (lower-case L) or **h**, indicating a **l**ong or **s**hort integer argument;
6. a *conversion character*, which indicates the type of data to be converted and printed.

As in *scanf*, a one-to-one correlation must exist between each specification encountered and each item in the item list.

The available *flags* are:

-	causes the data to be left-justified within its output field. Normally, the data is right-justified.
+	causes all signed data to begin with a sign (+ or −). Normally, only negative values have signs.
`blank`	causes a blank to be inserted before a positive signed value. This is used to line up positive and negative values in columnar data. Otherwise, the first digit of a positive value is lined up with the negative sign of a negative value. If the "blank" and "+" flags both appear, the "blank" flag is ignored.
#	causes the data to be printed in an "alternate form". Refer to the descriptions of the conversion characters below for details concerning the effects of this flag.

A *field width*, if specified, determines the *minimum* number of spaces allocated to the output field for the particular piece of data being printed. If the data happens to be smaller than the field width, the data is blank-padded on the left (or on the right, if the − flag is specified) to fill the field. If the data is larger than the field width, the field width is simply expanded to accommodate the data. An insufficient field width *never* causes data to be truncated. If no field width is specified, the resulting field is made just large enough to hold the data.

The *precision* is a value which means different things depending on the conversion character specified. Refer to the descriptions of the conversion characters below for more details.

Note: a field width or precision may be replaced by an asterisk (*). If so, the next item in the item list is fetched, and its value is used as the field width or precision. The item fetched must be an integer.

Conversion Characters

conversion character specifies the type of data to expect in the item list, and causes the data to be formatted and printed appropriately. The integer conversion characters are:

d an integer *item* is converted to signed decimal. The precision, if given, specifies the minimum number of digits to appear. If the value has fewer digits than that specified by the precision, the value is expanded with leading zeros. The default precision is one (1). A null string results if a zero value is printed with a zero precision. The # flag has no effect.

u an integer *item* is converted to unsigned decimal. The effects of the precision and the # flag are the same as for **d**.

o an integer *item* is converted to unsigned octal. The # flag, if specified, causes the precision to be expanded, and the octal value is printed with a leading zero (a C convention). The precision behaves the same as in **d** above, except that printing a zero value with a zero precision results in only the leading zero being printed, if the # flag is specified.

x an integer *item* is converted to hexadecimal. The letters **abcdef** are used in printing hexadecimal values. The # flag, if specified, causes the precision to be expanded, and the hexadecimal value is printed with a leading "0x" (a C convention). The precision behaves as in **d** above, except that printing a zero value with a zero precision results in only the leading "0x" being printed, if the # flag is specified.

X same as **x** above, except that the letters **ABCDEF** are used to print the hexadecimal value, and the # flag causes the value to be printed with a leading "0X".

The character conversion characters are as follows:

c the character specified by the **char** *item* is printed. The precision is meaningless, and the # flag has no effect.

s the string pointed to by the character pointer *item* is printed. If a precision is specified, characters from the string are printed until the number of characters indicated by the precision has been reached, or until a NULL character is encountered, whichever comes first. If the precision is omitted, all characters up to the first NULL character are printed. The # flag has no effect.

The floating-point conversion characters are:

f the **float** or **double** *item* is converted to decimal notation in *style f*; that is, in the form

[-]ddd.ddd

where the number of digits after the decimal point is equal to the precision. If no precision is specified, six (6) digits are printed after the decimal point. If the precision is explicitly zero, the decimal point is eliminated entirely. If the # flag is specified, a decimal point always appears, even if no digits follow the decimal point.

e the **float** or **double** *item* is converted to scientific notation in *style e*; that is, in the form

[−]d.ddd**Ae**±ddd

where there is always one digit before the decimal point. The number of digits after the decimal point is equal to the precision. If no precision is given, six (6) digits are printed after the decimal point. If the precision is explicitly zero, the decimal point is eliminated entirely. The exponent always contains exactly three digits. If the # flag is specified, the result always contains a decimal point, even if no digits follow the decimal point.

E same as **e** above, except that **E** is used to introduce the exponent instead of **e** (*style E*).

g the **float** or **double** *item* is converted to either *style f* or *style e*, depending on the size of the exponent. If the exponent resulting from the conversion is less than −4 or greater than the precision, *style e* is used. Otherwise, *style f* is used. The precision specifies the number of significant digits. Trailing zeros are removed from the result, and a decimal point appears only if it is followed by a digit. If the # flag is specified, the result always has a decimal point, even if no digits follow the decimal point, and trailing zeros are *not* removed.

G same as the **g** conversion above, except that *style E* is used instead of *style e*.

The *items* in the item list may be variable names or expressions. Note that, with the exception of the **s** conversion, pointers are **not** required in the item list (contrast this with *scanf*'s item list). If the **s** conversion is used, a pointer to a character string must be specified.

Examples

Here are some examples of *printf* conversion specifications and a brief description of what they do:

%d output a signed decimal integer. The field width is just large enough to hold the value.

%−*d output a signed decimal integer. The left-justify flag (−) and the blank flag are specified. The asterisk causes a field width value to be extracted from the item list. Thus, the item specifying the desired field width must occur before the item containing the value to be converted by the **d** conversion character.

%+7.2f output a floating-point value. The **+** flag causes the value to have an initial sign (+ or −). The value is right-justified in a 7-column field, and has exactly two digits after the decimal point. This conversion specification is ideal for a debit/credit column on a finance worksheet. (If the + sign is not necessary, use the blank flag instead.)

Consider the following program, which reads a number from **stdin**, and prints that number, followed by its square and its cube:

```
#include <stdio.h>
main()
{
   double x;

   printf("Enter your number: ");
   scanf("%F", &x);
   printf("Your number is %g\n", x);
   printf("Its square is %g\nIts cube is %g\n", x*x, x*x*x);
}
```

The **g** conversion character is used so that the decision about whether or not to use an exponent is automated. Note that the item list contains expressions to calculate *x* squared and *x* cubed. Also note that the address of the variable is required in order to read a value for it, but printing requires the variable name itself.

How about a program that accepts a decimal integer, and then prints the integer itself, its square, and its cube in decimal, octal, and hexadecimal? Easy enough:

```
#include <stdio.h>
main()
{
   long n, n2, n3;

/* get value */

   printf("Enter your number: ");
   scanf("%D", &n);

/* print headings */

    printf("\n\n              Decimal     Octal     Hexadecimal\n");

 /* do the computation */

   n2 = n * n;
   n3 = n * n * n;
   printf("n itself:    %7ld   %9lo     %6lx\n", n, n, n);
   printf("n squared:   %7ld   %9lo     %6lx\n", n2, n2, n2);
   printf("n cubed:     %7ld   %9lo     %6lx\n", n3, n3, n3);
}
```

This program prints the headings "Decimal", "Octal", and "Hexadecimal", and then prints out the data in tabular form. Programs which print tabular data always require some tinkering with the formats to make things come out right. Type this in and try it yourself.

Strings are especially easy to manipulate using *printf*. The following simple program illustrates this:

```
#include <stdio.h>
main()
{
   char first[15], last[25];

   printf("Enter your first and last names: ");
   scanf("%s%s", first, last);
   printf("\nWell, hello %s, it's good to meet you!\n", first);
   printf("%s, huh?  Are you any relation to that famous\n", last);
   printf("computer programmer, Mortimer Zigfelder %s?\n", last);
   printf("No, sorry, that was my mistake.  I was thinking of\n");
   printf("O'%s, not %s.\n", last, last);
}
```

This program shows how easily strings can be inserted in text. Try variations of your own.

Input/Output from/to Strings

Two library routines, *sscanf* and *sprintf*, enable you to read data from a string, and write data into a string. These routines behave identically to *scanf* and *printf*, respectively, except that *sscanf* reads data from a character string instead of from **stdin**, and *sprintf* writes data into a string instead of on **stdout**.

Reading Data from a String

Sscanf enables you to read data directly from a string. The syntax for an *sscanf* call is

```
sscanf(string, format, [item[, item ...]]);
```

where *string* is the name of a character array containing the data to be read, and *format* and *item* are familiar terms from the previous section. Thus, the only difference between *sscanf* and *scanf*, syntactically speaking, is *sscanf*'s inclusion of a new parameter, *string*.

The following program simply reads a string of your choosing from **stdin**, stores it in the character array *string*, and prints out the first word of that string:

```
#include <stdio.h>
main()
{
   char string[80], word[25], *gets();

/* get the string */

   printf("Enter your string: ");
   gets(string);

/* get the first word */

   sscanf(string, "%s", word);
   printf("The first word is %s.\n", word);
}
```

Of course, *sscanf* is rarely used in this way. *Sscanf* is more often used as a means of converting ASCII characters into other forms, such as integer or floating-point values. For example, the following program uses *sscanf* to implement a five-function calculator:

```
#include <stdio.h>
main()
{
   char line[80], *gets(), op[4];
   long n1, n2;
   double arg1, arg2;

/* print prompt (>) and get input */

   printf("\n> ");
   gets(line);

/* begin loop */

      while(line[0] != 'q') {
         sscanf(line, "%*s%s", op);
         if(op[0] == '+') {
            sscanf(line, "%F%*s%F", &arg1, &arg2);
            printf("Answer:  %g\n\n", arg1+arg2);
      } else if(op[0] == '|-') {
         sscanf(line, "%F%*s%F", &arg1, &arg2);
         printf("Answer:  %g\n\n", arg1|-arg2);
      } else if(op[0] == '*') {
         sscanf(line, "%F%*s%F", &arg1, &arg2);
         printf("Answer:  %g\n\n", arg1*arg2);
      } else if(op[0] == '/') {
         sscanf(line, "%F%*s%F", &arg1, &arg2);
         printf("Answer:  %g\n\n", arg1/arg2);
      } else if(op[0] == '%') {
         sscanf(line, "%D%*s%D", &n1, &n2);
         while(n1 >= n2)
            n1 |-= n2;
         printf("Answer:  %ld\n\n", n1);
      } else
         printf("Can't recognize operator: %s\n\n", op);
      printf("> ");
      gets(line);
   }
}
```

The calculator program accepts input lines having the form

value <operator> *value*

where *value* is any number, and <operator> is the symbol **+**, **−**, *****, **/**, or **%**, standing for addition, subtraction, multiplication, division, or remainder, respectively. All functions except remainder are handled internally in floating-point, but values for these functions can be typed with or without a decimal point. Values for the remainder function must not have a decimal point. There must be at least one space between each value and the operator.

Note the use of *sscanf* in this program. The entire input line is read using *gets*. Then, the different parts of the input line are read from *line* using *sscanf*. Notice that the input line is stored as an ASCII string in *line*, but portions of it are converted to floating-point or integer values, depending on the operator.

Examples of valid entries are

```
15.778 * 3.89
27 % 8
17 + 39.72
etc.
```

The program terminates when it reads a line beginning with "q", such as "quit".

There are two things that differ between reading data from **stdin**, and reading data from a string. First, you remember that reading data from **stdin** causes that data to "go away" −− it is no longer contained in **stdin**. This is *not* true for a string. Since the data is stored in a string, it is always there, even if that data has been read several times. Second, since the data read from **stdin** disappears as you read it, the next read operation from **stdin** always begins where the previous read operation terminated. This is *not* true when you read from a string using *sscanf*. Each successive read operation begins *at the beginning of the string*. Thus, if you want to read five words from a string stored in a character array, you must read them in a single *sscanf* call. If you try to read one word in five separate *sscanf* calls, each call starts reading at the beginning of the string, and you end up reading the same word five times!

Writing Data Into a String

The *sprintf* routine enables you to write data into a character string. Its syntax is

```
sprintf(string, format, [item[, item ...]]);
```

which is identical to that of *sscanf*. *String* is the name of the character string into which the data is written. *Format* and *item* are familiar terms from the previous discussion of *printf*. In fact, the only difference between *sprintf* and *printf* is that *sprintf* writes data into a character array, while *printf* writes data on **stdout**.

The following program acts as a "formatter" for personal data. Suppose that this program is used to provide a "friendly" user interface to gather personal data. The data received is then reformatted into a string which is passed along to another program, such as a data base maintainer. The string contains the data entered by the user, but in a form using strict field widths for the various pieces of data. The data base program requires these field widths in order for the data to be processed correctly, but there is no reason to burden the user with this requirement. This "formatter" program lets the user enter data in a convenient form (without the fixed field restrictions imposed by the data base).

```
#include <stdio.h>
main()
{
   char name[31], prof[31], hdate[7], curve[3], string[81];
   char *format = "%30s%2d%30s%61d%6s%2d%2s";
   int age, rank;
   long salary;

/* start asking questions */

   printf("\nName (30 chars max): ");
   gets(name);
   while(name[0] != ']') {
      printf("Age: ");
      scanf("%d%*c", &age);
      printf("Job title (30 chars max): ");
      gets(prof);
      printf("Salary (6 digits max, no comma): ");
      scanf("%D%*c", &salary);
      printf("Hire date (numerical MMDDYY): ");
      gets(hdate);
      printf("Percentile ranking (omit \"%%\"): ");
      scanf("%d%*c", &rank);
      printf("Pay curve: ");
      gets(curve);
```

```
/* format string */

        sprintf(string,format,name,age,prof,salary,hdate,rank,curve);
        printf("\n%s\n", string);

/* start next round */

        printf("\nName (30 chars max): ");
        gets(name);
    }
}
```

This program asks you questions to obtain typical company information such as name, age, job title, salary, hire date, ranking, and pay curve. This data is then packed into a 78-character string using *sprintf*. The string is printed on your screen in this program, but in an actual working environment, this string would probably be passed directly to the data base program. Note that *sprintf*'s format is specified as an explicit character pointer. When lengthy, unchanging formats are used, this is often more convenient than typing the entire format string, especially if the item list is long.

As an exercise, consider the *scanf* calls in the previous program. Notice that a **%*c** conversion specification is included in the formats of the *scanf*s which are reading integer values (age, salary, rank). Why is this necessary? If you aren't sure, take the **%*c**'s out of those formats, re-compile the program, run it, and note its behavior. (Hint: remember that a new-line character terminates the read operation for **%d** and **%D** conversions, and leaves the new-line unread in **stdin**.)

Input/Output Using Ordinary Files

So far, you have been using library routines which can perform I/O only by using **stdin** and **stdout**. This section introduces routines that enable you to open existing ordinary files for reading, writing, or both, and to create ordinary files. Routines that enable you to perform I/O to and from ordinary files are also described.

Opening Ordinary Files

Before a file can be read from or written to, it must be **opened**. A file is opened using the *fopen* library routine. The syntax of an *fopen* call is

```
fopen(<filename>, <type>);
```

where <filename> is a character pointer to a character string specifying the name of the file to be opened, and <type> is a character pointer to a one- or two-character string specifying the I/O operation for which the file is opened. The available <type>s are:

r	opens the file for reading at the beginning of the file. The file must already exist, or an error occurs.
w	opens the file for writing at the beginning of the file. If the file exists, its previous contents are destroyed. If the file does not exist, it is created.
a	opens the file for writing at the end of the file (appends data to the end of the file). If the file does not exist, it is created for writing.
r+	opens the file for both reading and writing, starting at the beginning of the file. The file must already exist, or an error occurs.
w+	opens the file for both reading and writing, starting at the beginning of the file. If the file already exists, its previous contents are destroyed. If the file does not exist, it is created.
a+	opens the file for both reading and writing, starting at the end of the file. If the file does not exist, it is created.

When a file is opened for an append operation (<type> is "a" or "a+"), it is impossible to overwrite the existing file contents. *Fseek* can be used to reposition the file pointer to any position in the file, but when output is written to the file, the pointer is disregarded. When the append operation (which begins at the end of the existing file) is completed, the file pointer is repositioned to the end of the appended output.

In exchange for a *filename* and a *type*, *fopen* opens a "pathway" between your program and the file. This "pathway" is called a *stream*. If you open the file for reading, then the stream provides one-way data transfer from the file to your program. If you open the file for writing, then data transfer flows from your program to the file. Finally, if the file is opened for both reading and writing, the resulting stream is bi-directional.

Fopen also associates a *buffer* with the stream. This gives the stream the ability to store a small amount of data. By default, the capacity of the buffer is equal to **BUFSIZ** bytes, where BUFSIZ is a constant defined in **stdio.h**. For the Series 200 and Series 500 computers, BUFSIZ is defined to be 1024.

The buffer size can be increased, decreased, or set to zero by using *setbuf* or *setvbuf*. If the buffer size is allowed to remain at default size, a maximum of BUFSIZ bytes of data can be present on the stream at any given time. If the buffer size is reduced to zero, then the stream can transfer only one byte at a time.

Since *fopen* takes care of all the intricacies of building a stream and allocating a buffer, all you need to know is how to find your end of the stream. *Fopen* provides you with this information by returning to you a value called a *file pointer* (often called a *stream pointer*). A file pointer "points" to the newly-created stream, and keeps track of where the next I/O operation takes place (in the form of a byte offset relative to the beginning of the associated buffer).

Is all this talk about streams and data transfer from a source to a destination beginning to sound familiar? Do you remember the "pipeline and water" analogy given at the beginning of this section? These two discussions should sound almost identical, because **stdin**, **stdout**, and **stderr** are actually file pointers to pre-opened streams! **Stdin** is a file pointer to a stream which transfers data from your tty (terminal) file to your program. **Stdout** and **stderr** are file pointers to two *different* streams which both transfer data from your program to your tty file. Be sure to note that **stdout** and **stderr** are different streams flowing in the same direction between the same two points!

Once you have a file pointer in your possession, you need never refer to the open file by its name again. A file pointer provides access to all the information needed by other standard I/O routines to read from or write to the file.

The following program fragment shows how the *fopen* routine is used:

```
#include <stdio.h>
main()
{
   FILE *fp;

   fp = fopen("/users/tom/bin/datafile", "r");
   if(fp == NULL) {
      printf("Can't open datafile.\n");
      exit(1);
   }
   ...
}
```

This *fopen* call, if successful, opens */users/tom/bin/datafile* for reading. The file pointer returned by *fopen* is stored in *fp*. Note that *fp*'s value is checked to see if it is NULL. This is because *fopen* returns a NULL pointer if the indicated file cannot be opened. It is good practice to check the value of a file pointer —— this is the only error indication facility that *fopen* provides.

The previous example also introduces a new type declaration, **FILE**. The FILE declaration is defined in **stdio.h**. In the example above, it defines *fp* as a variable containing a file pointer. Note that explicit declarations of functions returning file pointers is unneccessary —— **stdio.h** declares all such functions for you.

Before moving on, keep in mind that several things can stop you from successfully opening a file. First, HP-UX limits the number of files simultaneously open in a process (refer to the System Administrator Manual supplied with your system to find your system's limit). Remember that **stdin**, **stdout**, and **stderr** are automatically opened for you, so the maximum you can explicitly open is three fewer than the system limit. Second, you must have permission to open the file for the particular *type* you have specified (this permission is granted or denied by the file's mode). Third, trying to open a non-existent file using *type* **r** or **r+** always fails. Fourth, if the *filename* is specified incorrectly, contains a non-existent directory name, or contains an intermediate component which is not a directory, the open fails. This is not a complete list, but it contains some of the common reasons why an attempt to open a file might fail.

Single-Character Input/Output

Now that you know how to open files and obtain file pointers, you have a whole new set of I/O routines at your disposal, enabling you to perform all kinds of I/O operations. In fact, there are about three times as many available routines that utilize file pointers as there are routines that are limited to **stdin** and **stdout** only!

In this section, only those routines that read or write one character at a time are discussed. These routines are *getc*, *putc*, *fgetc*, and *fputc*. *Getc* and *putc* are macros defined in **stdio.h** which read one character from the specified stream, and write one character on the specified stream, respectively. They have the following syntax:

```
getc(stream);
putc(c, stream);
```

where *stream* is a file pointer obtained from *fopen*, and *c* is a variable of type **char** (or **int**) indicating the character to write on the indicated stream. A simple version of the HP-UX *cat* command can be written using these routines:

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
   int c;
   FILE *fp;

   if(argc != 2) {
      printf("Usage:  cat file\n");
      exit(1);
   }

   fp = fopen(argv[1], "r");
   if(fp == NULL) {
      printf("Can't open %s.\n", argv[1]);
      exit(1);
   }

   while((c = getc(fp)) != EOF)
      putc(c, stdout);
   putc('\n', stdout);

   exit(0);
}
```

This program accepts a single argument which is assumed to be the name of a file whose contents are to be printed on the user's terminal. The specified file is opened for reading, and the resulting file pointer *fp* is used in *getc* to read a character from the file. Each character read is written on **stdout** using *putc* (note that **stdout**, as well as **stdin** and **stderr**, are perfectly legal file pointers). The reading and writing loop is terminated when the constant EOF is returned from *getc*, indicating that the end of the file has been reached. This constant is defined in **stdio.h**.

Note that *getc* and *putc* can be made to behave exactly like the *getchar* and *putchar* routines discussed earlier by specifying the appropriate file pointer. In other words,

```
getc(stdin);
```

is identical to

```
getchar();
```

and

```
putc(c, stdout);
```

is identical to

```
putchar(c);
```

Thus, the *putc* call in the previous program could just as easily have been

```
putchar(c);
```

without altering the behavior of the program. However, if the destination of the data is somewhere other than the user's terminal, the flexibility of *putc* is required. Take, for example, the following program, which is a simple version of the HP-UX *cp* command:

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
   int c;
   FILE *from, *to;

   if(argc != 3) {
      printf("Usage:  cp fromfile tofile\n");
      exit(1);
   }

   from = fopen(argv[1], "r");
```

```
    if(from == NULL) {
        printf("Can't open %s.\n", argv[1]);
        exit(1);
    }
    to = fopen(argv[2], "w");
    if(to == NULL) {
        printf("Can't create %s.\n", argv[2]);
        exit(1);
    }

    while((c = getc(from)) != EOF)
        putc(c, to);

    exit(0);
}
```

This program accepts two arguments. The first is the name of the file to be copied, and the second is the name of the file to be created. The first file is opened for reading, and the second file is created for writing. The data from the first file is then copied directly to the newly-created file.

The *fgetc* and *fputc* routines are actual functions, not macros. Their syntax and usage is identical to that of *getc* and *putc*, so no examples are given here illustrating their use. However, here are some distinctions between the macro and function versions of these routines to help you decide which to use:

- A function call takes time, since the function call still exists at run-time. A macro call, however, takes no time at all, because the macro call is replaced with the actual code making up the macro during compilation, before run-time. Thus, generally speaking, programs containing macros run faster than programs containing the equivalent function calls.
- A function's code is localized in one section of the program. Each function call causes a jump to that section to execute the function. A macro call, however, is replaced with its code everywhere that macro call appears. Thus, programs containing macro calls generally require more space than programs containing the equivalent function calls.
- You may take the address of a function, and pass it as an argument. You cannot do this with a macro.

Given these guidelines, decide which routines to use based on your own constraints.

Character Push-Back

The *ungetc* routine enables you to push back a single character onto an input stream. This character is then returned by the next *getc* call (or equivalent).

Ungetc's syntax is as follows:

```
ungetc(c, stream);
```

where *c* is the character to be pushed back, and *stream* is the input stream where the push-back is to occur. Note that *c must* be a character that has been previously read from *stream*.

The following program simply reads one character from **stdin**, pushes it back onto **stdin**, re-reads the character, and checks to make sure that this character and the character originally pushed back are the same. A message is printed on **stdout** stating the outcome of the comparison.

```
#include <stdio.h>
main()
{
   int c1, c2;

   c1 = getchar();
   ungetc(c1, stdin);
   c2 = getchar();
   if(c1 == c2)
      printf("They're the same!\n");
   else

      printf("Oops!  They're different!\n");
}
```

One character's worth of push-back is guaranteed as long as something has been read from the stream prior to the push-back attempt, and provided that the stream is buffered. More characters could possibly be pushed back, but determining exactly how many characters of push-back you can safely perform is quite possibly not worth the effort. However, for completeness, the following statement is included as a method for determining the number of characters of push-back available at any given time:

```
numpb = ftell(stream) % BUFSIZ + 1;
```

where *ftell* is a function discussed in a later section, *stream* is a file pointer, and BUFSIZ is a constant defined in **stdio.h** containing the size of the buffer in bytes. After execution, *numpb* contains the number of characters of push-back available at that time.

String Input/Output

The *fgets* and *fputs* routines enable you to read or write strings from or to specified streams. Their syntax is as follows:

```
fgets(string, n, stream);
fputs(string, stream);
```

where *string* is a pointer to a character string, and *stream* is a file pointer to the input or output stream.

Fgets reads a character string from the specified *stream*, and stores it in the character array pointed to by *string*. *Fgets* reads $n-1$ characters, or up to a new-line character, whichever comes first. If a new-line character is encountered, it is retained as part of the string (contrast this with *gets*, which replaces the new-line with a NULL character). *Fgets* appends a NULL character to the string.

Fputs writes the character string pointed to by *string* on the specified *stream*, stopping when a NULL character is encountered. *Fputs* does *not* append a new-line character to the string when it is written. This is because *fputs* is intended for use with *fgets*, which incorporates a new-line character into the string if a new-line is encountered in the input.

The *cp* program written earlier can be re-written using *fgets* and *fputs*:

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
   char c, line[256], *fgets();
   FILE *from, *to;

   if(argc != 3) {
      printf("Usage:  cp fromfile tofile\n");
      exit(1);
   }

   from = fopen(argv[1], "r");
   if(from == NULL) {
      printf("Can't open %s.\n", argv[1]);
      exit(1);
   }
   to = fopen(argv[2], "w");
   if(to == NULL) {
      printf("Can't create %s.\n", argv[2]);
      exit(1);
   }
```

```
    while(fgets(line, 256, from) != NULL)
        fputs(line, to);

    exit(0);
}
```

This program functions exactly like the previous version of *cp* above. Note that *fgets*'s return value is compared to NULL in the **while** loop, since *fgets* returns the NULL pointer when it reaches the end of its input.

This program can easily be converted to a simple *cat* command. It only requires four changes. Can you see what they are? First, change the *argc* comparison such that it reads

```
if(argc != 2) . . .
```

(You might also want to change the associated usage message!) Second, remove the *to* file pointer, since you don't need it anymore. Third, remove the block of code which uses *fopen* to open the new file, and assigns a value to *to*. Fourth, change the *fputs* call such that it reads

```
fputs(line, stdout);
```

Here's the new *cat* command:

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
    char c, line[256], *fgets();
    FILE *from;

    if(argc < 2) {
        printf("Usage:  cat file\n");
        exit(1);
    }

    from = fopen(argv[1], "r");
    if(from == NULL) {
        printf("Can't open %s.\n", argv[1]);
        exit(1);
    }

    while(fgets(line, 256, from) != NULL)
        fputs(line, stdout);

    exit(0);
}
```

Formatted Input/Output

Just as there are versions of *scanf* and *printf* which perform string I/O, so there are versions which enable I/O using files. *Fscanf* enables you to read data of all types from a specified stream, and *fprintf* provides the capability of writing data on a stream. Their syntax is as follows:

```
fscanf(stream, format, [item[, item ...]]);
fprintf(stream, format, [item[, item ...]]);
```

Stream is a file pointer to an open stream. *Format* and *item* should be familiar terms from previous discussions.

The following program illustrates the use of the *fscanf* and *fprintf* routines:

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
   int count = 0;
   FILE *file;

   if(argc != 2) {
      fprintf(stderr, "Usage:  wdcnt filename\n");
      exit(1);
   }

   file = fopen(argv[1], "r");
   if(file == NULL) {
      fprintf(stderr, "Can't open %s.\n", argv[1]);
      exit(1);
   }

   while(fscanf(file, "%*s") != EOF)
      count++;

   printf("Number of words found:  %d\n", count);

   exit(0);
}
```

This program, named *wdcnt* (for "word count"), counts the number of "words" in the file specified as its only argument. A word is defined as a string of non-space characters.

Note how *fprintf* is used in this program. You learned in a prior discussion that **stderr** is typically used to output error messages or warning statements. In this program, *fprintf* is used to direct error messages to **stderr**. You don't lose anything by doing this, since data written on **stderr** appears on your terminal by default. However, you gain some important flexibility. Now that error output is written on a different stream than normal output, the error output (or the normal output) can be *redirected* to another destination. For example, invoking the previous program as

```
$ wdcnt <file1> 2>errmsgs
```

causes all output arising from erroneous conditions to be collected in the file **errmsgs**. For the *wdcnt* program, this is somewhat trivial, since the program terminates upon any error. However, for programs which output any number of warnings without terminating, this is a very useful capability. Not only does it keep normal, desired output from getting cluttered up with error messages, but it enables you to save output for later examination at your leisure. Thus, it is good programming practice to write error messages and warnings on **stderr**, and use **stdout** (or whatever your destination file is) to output normal data.

Binary Input/Output

The routines described in this section deal with data in its binary form — that is, the data is never converted to ASCII for user viewing. These routines are used to transfer raw data between two points, such as from a variable to a data file, or vice versa.

Two routines, *getw* and *putw*, are used to read or write an integer word (an **int**) to or from a stream, respectively. Their syntax is as follows:

```
getw(stream);
putw(w, stream);
```

where *stream* is a file pointer to the input or output stream, and *w* is the integer word to be output by *putw*.

The following program "sorts" a data file which has presumably been created earlier, and contains raw integer data. The program divides this data file into two new data files, one containing integer data whose absolute value is less than or equal to 32767, the other containing data whose absolute value is larger than 32767.

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
   int word;
   FILE *dfile, *datale, *datagt;

   if(argc != 2) {
      fprintf(stderr, "usage:  intsort filename\n");
      exit(1);
   }

   dfile = fopen(argv[1], "r");
   if(dfile == NULL) {
      fprintf("Can't open %s.\n", argv[1]);
      exit(1);
   }

   datale = fopen("dfle", "w");
   if(datale == NULL) {
      fprintf("Can't create dfle file.\n");
      exit(1);
   }

   datagt = fopen("dfgt", "w");
   if(datagt == NULL) {
      fprintf("Can't create dfgt file.\n");
      exit(1);
   }

   while((word = getw(dfile)) != EOF) {
      if(word <= 32767 && word >= -32767)
         putw(word, datale);
      else
         putw(word, datagt);
   }

   exit(0);
}
```

This program reads a word from the specified data file. If its absolute value is less than or equal to 32767, the word is written on a file called **dfle** in the user's current directory. Otherwise, the word is written on a file called **dfgt** in the current directory.

Note that this program works only on machines that use four-byte integers. Also, the comparison between *word* and the constant EOF is faulty, since EOF is defined to be −1, a valid integer. The section entitled *Stream Status Inquiry Routines* describes standard I/O routines which fix this problem.

Both of these routines transfer four bytes at a time. Again, there is no ASCII conversion associated with these routines, so if you attempt to print the contents of a file containing integer data output by *putw*, you will get garbage. Note that it makes little sense to input binary data from **stdin**, as in

```
getw(stdin);
```

unless **stdin** is redirected from a file containing binary data. Using *getw* to read data from your keyboard is futile. If you type in a valid-looking integer, like "1728", *getw* reads the ASCII values of those characters and stores them as an integer. This results in data being read which is *very* different from what you probably intended.

Two other routines, called *fread* and *fwrite*, provide much more flexible binary data input and output. Their syntax is as follows:

```
fread((char *)ptr, sizeof(*ptr), nitems, stream);
fwrite((char *)ptr, sizeof(*ptr), nitems, stream);
```

where *ptr* is a pointer to the beginning of a block (array) of data. This argument is cast as a character pointer because these routines expect a pointer of this type. The second argument specifies the number of bytes per *unit* of data (four bytes per **int**, one byte per **char**, *x* bytes per **struct**, etc.). The C operator **sizeof** is usually used to obtain this value. The third argument, *nitems*, is an integer specifying the number of units of data to read or write. For example, if *ptr* points to the beginning of a structure, **sizeof**(*ptr*) tells how many bytes make up that structure, and *nitems* tells how many structures to read. Actually, the second and third arguments above may be reversed in the argument list with no ill effects, because internally these routines simply multiply the two integers together to obtain the total number of bytes to read. Finally, *stream* is a file pointer to the input or output stream.

As an example, suppose you have a program which keeps track of certain employee data. Each employee is to be described in a single structure. Here is a simple program to do that:

```
#include <stdio.h>
struct emp {
          char       name[40]; /* name */
          char       job[40];  /* job title */
          long       salary;   /* salary */
          char       hire[6]   /* hire date */
          char       curve[2]  /* pay curve */
          int        rank;     /* percentile ranking */
}
#define EMPS 400               /* no.  of employees */
main()
{
     int items;
     struct emp staff[EMPS];
     FILE *data;

     data = fopen("/usr/lib/employees/empdata", "r");
     if(data == NULL) {
        fprintf(stderr, "Can't open employee data file.\n");
        exit(1);
     }

     items = fread((char *)staff, sizeof(staff[0]), EMPS, data);
     if(items != EMPS) {
        fprintf(stderr, "Insufficient data found.\n");
        exit(1);
     }

     fclose(data);
     archive("/usr/lib/employees/empdata");

/* Employee information processing goes here. */

     ...

/* Processing is done.  Write out new employee records. */

     data = fopen("/usr/lib/employees/empdata", "w");
     if(data == NULL) {
        fprintf(stderr, "Can't create new employee file.\n");
        exit(1);
     }
```

```
        items = fwrite((char *)staff, sizeof(staff[0]), EMPS, data);
        if(items != EMPS) {
            fprintf(stderr, "Write error!\n");
            exit(1);
        }

        exit(0);
}
archive(filename)
char *filename;
{
        ...
}
```

This program reads the employee information contained in the binary file */usr/lib/employees/empdata*. The data in this file consists of concatenated streams of bytes describing each employee of a certain 400-employee company. The bytes are written such that, when read correctly, the bytes correspond exactly with the **emp** structure defined in the program. The *staff* array is an array of structures containing one structure for each employee.

In the *fread* call, the **sizeof**(*staff*[0]) expression returns the number of bytes in the **emp** structure. Since the same number of bytes are in each employee structure, any element of the *staff* array could have been specified as the **sizeof** argument; *staff*[0] is used in this example. (By counting the number of bytes in each structure member, you can get an approximation of the number of bytes returned by the **sizeof** operator: 40 + 40 + 8 + 6 + 2 + 4 = 100 bytes. This may vary due to padding performed by a programming language, or by machine architecture.) Specifying EMPS as the *nitems* argument tells *fread* to read 400 such structures. Thus, 100 x 400 = 40000 bytes are read, filling in the information for the members of each structure contained in the *staff* array.

The *archive* function is not shown here, but simply saves the old employee information in **empdata** in an employee information archive of some kind. After the information is archived, the **empdata** file is overwritten with the new, updated employee information.

A new routine, called *fclose*, is introduced here. *Fclose* simply closes the stream associated with the file pointer specified. This is necessary in order to re-open the file for writing. Once it is open for writing, *fwrite* is used to overwrite its previous contents with the new data.

One final note about these two routines: they return the number of items of data which have been read or written. Thus, you can compare this number with whatever you specified for *nitems* to see if everything you wanted read or written actually was. This return value is used twice in the above program to flag probable read and write errors.

The *fread* and *fwrite* routines can be made to read *any* type of data. The following examples show some *fread* calls which read several different types of data:

To read a long integer:

```
long nint;
fread((char *)&nint, sizeof(nint), 1, stream);
```

To read an array of 100 long integers:

```
long nint[100];
fread((char *)nint, sizeof(nint[0]), 100, stream);
```

To read a double precision floating-point value:

```
double fpoint;
fread((char *)&fpoint, sizeof(fpoint), 1, stream);
```

To read an array of 50 floating-point values:

```
float fpoint[50];
fread((char *)fpoint, sizeof(fpoint[0]), 50, stream);
```

To get the equivalent *fwrite* calls, just substitute "fwrite" in place of "fread" in the previous examples. You can see how much more flexible *fread* and *fwrite* are than *getw* and *putw*. Whereas *getw* and *putw* are limited to reading or writing a single four-byte integer per call, *fread* and *fwrite* can be made to read or write any number of variables of any type.

Stream Status and Control Routines

This section discusses standard I/O routines which enable you to:

- Determine whether or not an error has occurred on an open stream (*feof*, *ferror*, *clearerr*);
- Re-position the location of the next I/O operation on an open stream (*rewind*, *ftell*, *fseek*);
- Control various attributes of an open stream, such as buffering, flushing, etc. (*fclose*, *setbuf*, *fflush*, *freopen*);
- Convert a file pointer to a file descriptor, and vice versa (*fileno*, *fdopen*).

Stream Status Inquiry Routines

This section describes three routines, *feof*, *ferror*, and *clearerr*, which enable you to determine the status of an open stream at any given time.

Feof is a macro defined in **stdio.h** which returns a non-zero value if the end-of-file has been reached on an input stream. Its syntax is as follows:

```
feof(stream);
```

Do you remember the example program which illustrated the use of *getw* and *putw*? It was noted that comparing *getw*'s return value to the constant EOF was faulty, because *getw* returns an integer, and EOF is defined to be a valid integer (−1). How then do you determine if end-of-file has been reached when routines like *getw* are being used? You use *feof*.

The example program for *getw/putw* can be changed to use *feof*:

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
   int word;
   FILE *dfile, *datale, *datagt;

   if(argc != 2) {
      fprintf(stderr, "usage:  intsort filename\n");
      exit(1);
   }

   dfile = fopen(argv[1], "r");
   if(dfile == NULL) {
      fprintf("Can't open %s.\n", argv[1]);
      exit(1);
   }

   datale = fopen("dfle", "w");
   if(datale == NULL) {
      fprintf("Can't create dfle file.\n");
      exit(1);
   }

   datagt = fopen("dfgt", "w");
   if(datagt == NULL) {
      fprintf("Can't create dfgt file.\n");
      exit(1);
   }

   for(;;) {
      if((word = getw(dfile)) != EOF) {
         if(word <= 32767 && word >= |-32767)
            putw(word, datale);
         else
            putw(word, datagt);
       } else {
            if(feof(dfile))
               break;
            else
               putw(word, datale);
       }
   }

   exit(0);
}
```

An infinite loop is set up around the *getw/putw* process. Whenever *getw* returns an integer equal to EOF, *feof* is used to find out if end-of-file has been reached. If it has, the loop (and the program) terminates; if not, the integer is written on **dfle**, and the loop continues.

Ferror is a routine which examines the specified stream to determine whether or not a read or write error has occurred. Its syntax is

```
ferror(stream);
```

Ferror, like *feof*, is intended to clarify ambiguous return values from standard I/O routines. Actually, only *getw* and *putw* require the use of *ferror* to determine if an error has occurred. Both of these routines return EOF on end-of-file or error. Since these routines deal with integer data, however, you need *feof* and *ferror* to determine if the EOF returned actually indicated an error or an end-of-file, or if it's just a -1.

If an error has occurred on a stream, *ferror* returns a non-zero value.

Whenever an error occurs on an open stream, a flag is set to indicate the error. It is this flag that *ferror* checks to determine whether or not an error has occurred. This flag is *not* reset when it is checked. Thus, if an error has occurred, the error flag for that stream remains set. This could lead to misleading information if an *ferror* call indicates that an error has occurred, when in reality the error occurred long ago. The *clearerr* routine clears (or resets) the error indication flag for the specified stream. This routine should be used whenever an error has been indicated, so that the same error is not indicated at a later time. *Clearerr*'s syntax is

```
clearerr(stream);
```

Because *ferror* and *clearerr* are used infrequently in typical programs, no examples are given specific to their use. The *feof* example above illustrates the general scenario in which all three of these routines are used.

Re-positioning Stream I/O Operations

There are three routines, *rewind*, *ftell*, and *fseek*, which enable you to move the location of the next I/O operation on an open stream.

Rewind simply positions the next I/O operation at the beginning of the file. Its syntax is

```
rewind(stream);
```

For example, suppose a particular application program can put a password on a data file it uses. This password is stored in encrypted form on the first line of the file. The line is recognized as a password line if the first two characters are "*P". If the file has no password line, then access to the file is unrestricted. If a password line is found, the user is prompted for the password before access is permitted. The following code can be used to look for a password line:

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
   FILE *pswd;
   char line[256];

   if(argc != 2) {
      fprintf(stderr, "Usage: getpswd file\n");
      exit(1);
   }

   pswd = fopen(argv[1], "r");
   if(pswd == NULL) {
      fprintf(stderr, "Can't open %s.\n", argv[1]);
      exit(1);
   }

   fgets(line, 256, pswd);
   if(line[0] == '*' && line[1] == 'P') {

   /* ask for and check password */

   } else
      rewind(pswd);

     ...    /* application program goes here */

   exit(0);
}
```

If the first two characters of the first line are "*P", then code is executed which asks for and checks a password. However, if the first line is not a password line, the file is assumed to be unprotected, and the line just read is probably part of the data. Thus, the file must be rewound so the data contained in the first line is available to the application program.

The *ftell* routine returns a long integer specifying the current position of the next I/O operation on an open stream. This position is expressed as a byte offset relative to the beginning of the open file. Its syntax is as follows:

```
ftell(stream);
```

The *fseek* routine enables you to re-position the next I/O operation on an open stream to any location you wish. Its syntax is

```
fseek(stream, offset, ptrname);
```

where *stream* is a file pointer to the open stream, *offset* is a long integer specifying the number of bytes to skip over, and *ptrname* is an integer indicating the reference point in the file from which *offset* bytes are measured. The possible values for *ptrname* are:

0 move *offset* bytes from the beginning of the file;

1 move *offset* bytes from the current position in the file;

2 move *offset* bytes from the end of the file.

Offset can be either negative or positive, indicating backward or forward movement in the file, respectively.

The following program illustrates the use of the *ftell* and *fseek* library routines. The program prints each line of an *n*-line file in this order: line 1, line *n*, line 2, line $n-1$, line 3, ...

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
   char line[256];
   int newlines;
   long front, rear, ftell();
   FILE *fp;

   front = 0;
   rear = 0;

   if(argc < 2) {
      fprintf(stderr, "Usage:  print filename\n");
```

```
        exit(1);
    }

    fp = fopen(argv[1], "r");
    if(fp == NULL) {
        fprintf(stderr, "Can't open %s.\n", argv[1]);
        exit(1);
    }

    newlines = countnl(fp) % 2;

    fseek(fp, 0, 2);
    rear = ftell(fp);

    while(front < rear) {
         fseek(fp, front, 0);
         fgets(line, 256, fp);
         fputs(line, stdout);
         front = ftell(fp);
         findnl(fp, rear);
         rear = ftell(fp);
         if(newlines == 1) {
              if(rear <= front)
                   break;
         }
         fgets(line, 256, fp);
         fputs(line, stdout);
    }

    exit(0);
}

countnl(fp)
FILE *fp;
{
    char c;
    int count = 0;

    while((c = getc(fp)) != EOF) {

if(c == '\n')

    count++;
    }
    rewind(fp);
    return(count);
}

findnl(fp, offset)
FILE *fp;
```

```
long offset;
{
   char c;

   fseek(fp, (offset-2), 0);
   while((c = getc(fp)) != '\n') {

fseek(fp, -2, 1);
   }
}
```

This program uses *ftell* and *fseek* to print lines from a file starting at the beginning and the end of the file, and converging toward the center. The **countnl** (count new-lines) function counts the number of lines in the file so the program can decide whether or not to print a line in the final loop (this prevents the middle line being printed twice in files with an odd number of lines). The **findnl** (find new-line) function seeks backwards in the file for the next new-line. When found, this positions the next I/O operation such that *fgets* gets the next line back from the end of the file.

Note the use of *fseek* in this program. All three types of seeks are represented here. The first *fseek* of the program is done relative to the end of the file. All other *fseeks* in the main program are done relative to the beginning of the file. Finally, **findnl** contains an *fseek* which is relative to the current position.

Recall the employee data routine, where each employee is described by the structure

```
struct emp {
          char      name[40]; /* name */
          char      job[40];  /* job title */
          long      salary;   /* salary */
          char      hire[6];  /* hire date */
          char      curve[2]; /* pay curve */
          int       rank;     /* percentile ranking */
}
```

That routine simply read in the data for 400 employees all at once. Suppose you want the program to be selective, so that you can specify (by employee number, 1 – 400) *which* employee's information you want. This is easily done using *fseek*. The following program fragment shows how:

```
    ...

    int empno, bytes;
    long total;
    FILE *data;
    struct emp empinfo;

    /* check for usage error and open data file */
    ...

    sscanf(argv[1], "%d", &empno);
    bytes = sizeof(empinfo);
    total = (empno - 1) * bytes;
    fseek(data, total, 0);
    fread((char *)&empinfo, sizeof(empinfo), 1, data);

    /* print out desired information */
    ...

    exit(0);
}
```

In this program, argv[1] contains, via a command-line argument, the employee number about whom information is desired. This employee number is converted to integer form using *sscanf*. The number of bytes per employee structure is obtained using **sizeof**, and is stored in *bytes*. The total number of bytes to skip in the data file is found by multiplying the employee number (minus one) times the number of bytes per employee structure. This is stored in *total*. Then, *fseek* is used to seek past the specified number of bytes, relative to the beginning of the data file. This leaves the next I/O operation positioned at the start of the specified employee's information. The information is read using *fread*.

NOTE

If you have a stream which is open for both reading and writing, a read operation cannot be followed by a write operation without one of the following occurring first: a *rewind*, an *fseek*, or a read operation which encounters end-of-file. Similarly, a write operation cannot be followed by a read operation unless a *rewind* or *fseek* is performed.

Stream Control Routines

The routines described here help you control certain attributes of file pointers. The routines described are *fclose*, *setbuf*, *setvbuf*, *fflush*, and *freopen*.

fclose

You have already seen *fclose* in action in the previous example program which read an employee data file. *Fclose* flushes the buffer associated with the specified stream, and, if the buffer was allocated automatically by the standard I/O system, frees the space allocated to that buffer. The stream is then closed, breaking the connection between your file pointer and the stream.

You may be wondering why so many example programs have been written that open files but never explicitly close them. There are two reasons why this is permissible. First, you'll notice that all programs in this tutorial that open files end with a call to *exit*. The *exit* system call automatically performs an *fclose* for every open file in that process. Second, when a program is compiled with *cc* (or *fc*, or *pc*), an *exit* call is automatically compiled in with your code. Keep in mind, however, that it is generally bad programming practice to rely on the system to clean up after you! If you explicitly open any files, you should explicitly close them when you are done. If this is too much trouble, at least include an *exit* call at each termination point in the program. (All future example programs in this article will contain *fclose* calls.)

Setbuf

Setbuf and *setvbuf* routines enable you to assign your own buffering to an open stream. *Setbuf* syntax is

```
setbuf(stream, buffer);
```

where *stream* is a file pointer to an already-open stream, and *buffer* is a pointer to a character array or is NULL.

Normally (i.e. without user intervention), a standard I/O buffer is obtained through a call to *malloc*(3C) (*memallc*(2) on the Series 500) upon the first call to *getc* or *putc* (which all I/O routines eventually call). The standard I/O system normally buffers I/O in a buffer which is BUFSIZ bytes long. Exceptions are *Stdout*, which, when directed to a terminal, is line-buffered, and **stderr**, which is normally unbuffered.

Setbuf enables you to change the buffer used for all standard I/O routines. For example, the following code fragment causes the array *buffer* to be used for buffering:

```
...

FILE *fp;
char buffer[BUFSIZ];

fp = fopen(argv[1], "r");
...

setbuf(fp, buffer);
...
```

This fragment shows the correct order of events. First, the file is opened (it need not be opened for reading), then the buffering is assigned using *setbuf*. From that point on, any input taken from *fp* is buffered through the array *buffer*.

Buffering can be eliminated altogether by specifying the NULL pointer in place of the buffer name, as in

```
setbuf(fp, NULL);
```

This causes input or output using *fp* to be completely unbuffered.

Setbuf is limited to buffer sizes of either BUFSIZ bytes or zero. *Setbuf* assumes that the character array pointed to by "buffer"is BUFSIZ bytes. Passing *setbuf* a (non-NULL) pointer to a smaller array can cause severe problems during operation because the standard I/O routines may overwrite memory following the end of the too-small buffer.

Note: Using an *automatic* array as a standard I/O buffer can be dangerous. Automatic variables are only defined in the code block in which they are declared. Thus, buffering which relies on an automatic array is only in effect during the current code block (main program or function). If you pass a file pointer to another function, and the stream pointed to by that file pointer is buffered using an automatic array, then memory faults or other errors can occur. Here's the rule: if you use an automatic array for stream buffering, the stream should be used and closed only in the code block containing the array declaration. To avoid this restriction, use **external** arrays for buffering:

```
...
external char buffer[BUFSIZ];
...
setbuf(fp, buffer);
```

Setvbuf

Setvbuf, like *setbuf*, enables you to assign a character array for buffering, but also provides the means to specify the size of the buffer to be used and the type of buffering to be done. *Setvbuf* syntax is

```
setvbuf(stream, buffer, type, size)
```

where *stream* is a file pointer to an already-open stream, *buffer* is a pointer to a character array or is NULL, *type* tells how *stream* is to be buffered, and *size* defines how large the *buffer* is. Acceptable values for *type* (defined in stdio.h) include:

–IOFBF	Input/output is fully buffered.
–IOLBF	Output is line buffered. The buffer is flushed each time a new line is written, the buffer is full, or input is requested.
–IONBF	Input/output is completely unbuffered.

If *type* –IONBF is specified, *stream* is totally unbuffered. Since no buffer is needed, values for *buffer* and *size* are ignored. For example, the following two calls, though different, are functionally identical:

```
setvbuf(fp, NULL, -IONBF, 0)
setbuf(fp, NULL)
```

When *type* is –IOFBF or –IOLBF, buffering for *stream* is determined by *buffer* and *size*. If *buffer* is not the NULL pointer, it must point to a character array of *size* bytes. All buffering of *stream* is then handled through this array.

```
        ...
        FILE *fp;
        char buffer [256]
        char *filename;
        int ... retcode;
        fp=fopen(filename, "w");
retcode=setvbuf(fp, buffer, =IOFBF, 256);
        if (retcode !=0) error c);
```

This fragment causes stream *fp* to be buffered through the 256-byte array *buffer*. Serious run-time errors can occur if the buffer array is not the size specified in the call to *setvbuf* (here 256 bytes). As with *setbuf*, it is dangerous to use an automatic array for the buffer. Note that the return value of *setvbuf* can be used to verify that the request was completed successfully.

If *buffer* is the NULL pointer and type is specified as –IOFBF or –IOLBF, *setvbuf* automatically allocates a buffer of *size* bytes through a call to *malloc* (3c) on Series 200 computers or *memallc* (2) on Series 500 computers. If *size* is zero, a buffer of size BUFSIZ will be used. This behavior can be used to change the buffer size for a stream even if you still want the standard I/O system to automatically allocate the buffer. This is particularly useful when a buffer larger than the specified BUFSIZ is desired.

```
        ...
        FILE * fp;
        char * filename;
        int retcode;
        ...
        fp = fopen(filename, "rt")
retcode=setvbuf(fp, NULL, -IOFBF, 2048);
        if(retcode !=0) error( );
        ...
```

This fragment buffers stream *fp* through a 2048-byte buffer that is allocated by the system.

fflush

The *fflush* routine forces all buffered data for an output stream to be written out to that file. Its syntax is

```
fflush(stream);
```

where *stream* is a file pointer to an output stream.

Fflush is performed automatically by *fclose* (and, therefore, by *exit*). Therefore, there is often no reason to call *fflush* explicitly. Situations do arise, however, where it is necessary to manually *fflush* a stream. For example, data written to a terminal is line-buffered by default, which means that the system waits for a new-line before writing the buffer onto the terminal screen. This is often satisfactory, but there are times when you want whatever has been written so far to be written to the screen without waiting for the new-line. In such situations, *fflush* must be used.

Another situation when explicit *fflush*ing is necessary arises whenever you have written less than a buffer-full of data to a file, and you want the contents of that file processed by another function, or by an HP-UX command. Since less than a buffer-full of data was written, the data is still in the buffer; the file is still empty. Performing an *fflush* causes the buffered data to be written out to the file, enabling other functions or commands to utilize the file's contents.

freopen

The final routine in this section is *freopen*. As its name implies, *freopen* enables you to, in a single step, close a stream and then re-open it with a different type and/or file name. Its syntax is

```
freopen(filename, type, stream);
```

where *filename* is a pointer to a character string specifying the name of the source or destination file for the newly-created stream. *Type* is identical to that of *fopen* discussed earlier. *Stream* is a file pointer to the old stream, which is closed and then re-opened. The name of the file pointer remains the same.

For example, the following program accepts lines of data from your terminal and writes them into a file. When only a new-line is typed from the terminal, the program quits reading data, and echos the contents of the file to the terminal.

```
#include <stdio.h>
main()
{
  FILE *fp, *oldfp;
  char line[80], *fgets( );

  fp = fopen("datafile", "w");
  if(fp == NULL) {
     fprintf(stderr, "Can't create datafile.\n");
     exit(1);
  }

  fgets(line, 80, stdin);
  while(line[0] != "\n") {
     fputs(line, fp);
     fgets(line, 80, stdin);
  }

  oldfp = freopen("datafile", "r", fp);
  if(oldfp == NULL) {
     fprintf(stderr, "Can't re-open datafile.\n");
     exit(1);
  }

  while(fgets(line, 80, fp) != NULL)
     fputs(line, stdout);

  fclose(fp);
  exit(0);
}
```

Just like *fopen*, *freopen* returns a NULL pointer if an error occurs. If successful, *freopen* returns the value of the old file pointer.

Freopen is commonly used to attach the names **stdin**, **stdout**, and **stderr** to other files, so that the source or destination of these file pointers can be redirected. For example,

```
freopen("/usr/lib/data/datafile", "r", stdin);
```

attaches **stdin** to the data file */usr/lib/data/datafile*. Other functions can now be called which read from **stdin**, and the result is that their source of input has been redirected. Similarly,

```
freopen("/users/bill/archives/cal.a", "a", stdout);
```

attaches **stdout** to the indicated file, thus redirecting any future **stdout** data to that file.

Converting Between File Pointers and File Descriptors

A file pointer is actually a pointer to a structure containing information about a stream. This information includes a pointer to the beginning of the buffer, a pointer to the current location in the buffer, a flag specifying whether the stream is open for reading, writing, or both, a count of the characters in the buffer, and an integer called a *file descriptor*.

System calls, such as *open* and *creat*, return a file descriptor when a file is opened. System calls use file descriptors to refer to open files in much the same way that library routines use file pointers. (The main difference between using a file descriptor and using a file pointer is that a file descriptor has no associated buffering.) Since a program often contains both system calls and library routines, a way of converting between file pointers and file descriptors is provided.

NOTE

Extreme care should be exercised when converting between file pointers and file descriptors. Whenever you convert a file pointer to a file descriptor, you should perform an *fflush* first.

In general, you should never convert file pointers to file descriptors unless you need a file descriptor for a system call that provides a utility not available in the C library package (such as *dup*(2) or *fcntl*(2)). Similarly, file descriptors should never be converted to file pointers unless a file descriptor has been created by a system call which provides a utility not provided in the C library package, and you want to assign system buffering to it.

Two routines, *fileno* and *fdopen*, provide a way to convert between the two types of parameters. *Fileno* is a macro which, given a file pointer, returns the associated file descriptor. Its syntax is

```
fileno(stream);
```

where *stream* is a file pointer to an open stream whose associated file descriptor is desired. Thus,

```
...
FILE *fp;
int fd;
...
fp = fopen("file1", "r");
fd = fileno(fp);
```

returns the integer file descriptor in *fd*, associated with the file pointer *fp*.

The *fdopen* routine enables you to convert a file descriptor into a file pointer. Its syntax is

```
fdopen(fildes, type);
```

where *fildes* is an integer file descriptor obtained from the *open*, *dup*, *creat*, or *pipe* system calls. *Type* is the same as that for *fopen* discussed earlier. Thus,

```
...
int fd;
FILE *fp;
...
/* obtain fd via appropriate system call */
...
fp = fdopen(fd, "r");
if(fp == NULL) {
     fprintf(stderr, "Can't convert file descriptor.\n");
     exit(1);
}
...
```

converts the file descriptor *fd* into a file pointer, *fp*. *Fdopen* returns a NULL pointer if the operation fails.

Fdopen can be useful for opening a file in a way unlike any of the standard types of *fopen*.

```
...
# include <fcntl.h>
...
int fd;
FILE *fp
char *filename;
...
fd= open(filename, O_WRONLY|O_CREAT, 0666);
fp= fdopen(fd,"w");
fseek(fd,0L,2)
```

This code fragment uses the *open* system call to open a file for general write access, then uses *fdopen* to assign buffering to the file. The constants O_WRONLY and O_CREAT are defined in the include file */usr/include/fcntl.n*, and are described in *open* (2). (`O_WRONLY` causes *open* to open the file for writing only; `O-CREAT` creates the file if it does not already exist.) This technique opens the file in a way that does not correspond exactly to any of the available types in *fopen*: "w" would truncate the current file contents, "r+" would fail if the file does not already exist (and would allow reading of the file), and "a" does not permit seeking backwards and rewriting the current file contents.

Inter-Process Communication

So far, you've been communicating between an active process (your program) and a passive object (a file). What if you want to communicate between two active processes? Suppose you want to create a stream between two programs, with one program (process) pumping data onto the stream, and the other reading data from the other end. How is this done?

The *popen* routine exists for this purpose. Its syntax is

```
popen(command, type);
```

where *command* is a pointer to a character string specifying a command line. *Type* is a pointer to a single-character string which is either **r** (for reading) or **w** (for writing).

For example, suppose you are writing a program which processes text in some way. Your program handles normal text perfectly, but unfortunately your source files are all coded in *troff* constructs. If you could only filter out all those pesky *troff* constructs, your program would work fine. Cheer up! It's easily done. There is an HP-UX command called *deroff* which filters out *troff* constructs. All you have to do is make sure that all input to your program passes through *deroff* first. Here's how:

```
#include <stdio.h>
main()
{
   FILE *popen(), *fp;

   fp = popen("deroff /users/bin/text/*.tx", "r");
   if(fp == NULL) {
      fprintf(stderr, "Can't create stream.\n");
      exit(1);
   }

   /* begin processing text; read text from fp! */
   ...
   pclose(fp);
```

Popen returns a file pointer to the newly-opened stream. If an error occurs, a NULL pointer is returned. When successfully executed, *popen* enables your program to read from the file pointer *fp*, the data from which is the standard output from the *deroff* command. In this example, *deroff* is invoked such that it processes all files in */users/bin/text* which end with ".tx". Note that *popen*'s return value must be declared explicitly because it is not declared in **stdio.h**.

Because *deroff* processes **stdin** if no arguments are given, the following *popen* call

```
fp = popen("deroff", "r");
```

enables your program to receive filtered text from **stdin** instead of from ordinary files. The result of executing the previous example is exactly the same as if you had typed

```
deroff /users/bin/text/*.tx | yourprogram
```

at your keyboard in response to a shell prompt.

Streams that are opened by *popen* must be closed with *pclose*. Thus,

```
pclose(fp);
```

closes the stream created in the previous example.

If a *type* of **w** is specified instead of **r**, then the data flow is reversed, with the result that your program supplies the data for the specified *command*.

Note that, though *popen*'s return value is called a file pointer, it is actually somewhat different than the file pointers you are already familiar with. In general, a file pointer returned by *popen* should not be used in those previously-discussed library routines which modify file pointers returned by *fopen*. Also, file pointers opened by *popen* must be closed with *pclose*; *fclose* is not sufficient.

So far, *popen* has been characterized as a "filter-maker", in that streams to or from a command have been created so that data can be modified in some way before being passed on. Sometimes, however, *popen* is used to execute a command which supplies information valuable to the program. For example, the *find* command accepts dot (.) as a valid directory name. Upon receipt of a dot, *find* discovers the actual path name of dot by creating a stream from the *pwd* command, as follows:

```
char dir[100];
FILE *popen(), *fp;

fp = popen("pwd", "r");
if(fp == NULL) {
     fprintf(stderr, "Can't execute pwd.\n");
     exit(1);
}
fgets(dir, 100, fp);
...
pclose(fp);
```

The preceding example reads the output of the *pwd* command into the character array *dir*, thus supplying the current value of dot. The following program creates a list of the login names of users currently logged in:

```
#include <stdio.h>
main()
{
   char name[10], line[80], *fgets();
   FILE *popen(), *fp;

   fp = popen("who", "r");
   if(fp == NULL) {
      fprintf(stderr, "Can't execute who.\n");
      exit(1);
   }

   printf("Users currently logged in:\n");
   while(fgets(line, 80, fp) != NULL) {
      sscanf(line, "%s", name);
      printf("\t%s\n", name);
   }

   pclose(fp);
   exit(0);
}
```

A stream is created for reading from the *who* command. Each line from *who* is read, and the first field from each line is read and printed.

You may have only one *popen*-ed stream in a process at any given time.

Part 2:
Math Routines

Described in this section are absolute value, power, square root, logarithmic, trigonometric, and other functions performing many different kinds of mathmatical calculations.

An include file named **math.h** exists for use with these routines. **Math.h** contains type declarations of all the math routines which do not return an **int**, and a definition of the constant **HUGE**. Many math routines return a "huge" value when an error occurs, so HUGE is set equal to this "huge" value, enabling you to check for errors easily. You need not include **math.h** in your program **if** you remember to explicitly declare each math routine's return type, and **if** you don't need HUGE.

Some of the math routines reside in the standard C library, */lib/libc.a*. This library also contains all the standard I/O routines and the system calls described in section 2 of the HP-UX Reference manual. This library is loaded automatically by the C compiler, *cc*, so you need not worry about explicitly telling the linker (*ld*) to search this library to find the functions contained in it. However, many math routines reside in the library */lib/libm.a*, which is *not* automatically loaded. Thus, if you try to compile a program containing a math routine from *libm.a*, you get a complaint from *ld*.

This is fixed in the following way. Suppose you have a program named *yourprog.c*, and this program contains a math function from *libm.a*. To compile the program, type

```
$ cc yourprog.c -lm
```

The **−l** option causes *ld* to look for and search a library named /lib/libx.a, where *x* is the letter specified after the **−l** option. Thus, this command line tells *ld* to search /lib/libm.a.

How do you know which functions reside in which library? The HP-UX Reference manual provides guidance here. /lib/libc.a contains all of section 2, plus all routines in section 3 having the suffixes (3C) and (3S). /lib/libm.a contains all the routines in section 3 having the suffix (3M). To aid you in deciding how to compile your programs, the routines discussed below include references to the HP-UX Reference manual.

Absolute Value Functions

The *abs* (*abs*(3C)) and *fabs* (found under *floor*(3M)) functions return the absolute value of their integer or floating-point argument, respectively. For example, the following program calculates integer absolute values until a zero is entered from the keyboard:

```
main()
{
   int value;

   printf("Enter value: ");
   scanf("%d", &value);
   while(value != 0) {
        printf("Absolute value of %d is %d.\n", value, abs(value));
        printf("Enter value: ");
        scanf("%d", &value);
   }
   exit(0);
}
```

The floating-point equivalent of the previous program is shown below:

```
main()
{
   double value, fabs();

   printf("Enter value: ");
   scanf("%lf", &value);
   while(value != 0.0) {
        printf("Absolute value of %.12g is %.12g.\n", value, fabs(value));
        printf("Enter value: ");
        scanf("%lf", &value);
   }
   exit(0);
}
```

The first program above can be compiled without the **−l** option, but the second must be compiled using the **−lm** option.

Power, Square Root, and Logarithmic Functions

This section describes the following five functions, all of which are found under *exp*(3M) in the HP-UX Reference manual:

exp(x)	returns e to the *x* power.
log(x)	returns the natural logarithm of *x* (ln(x)).
log10(x)	returns the common logarithm of *x* (log(x)).
pow(x, y)	returns *x* to the *y* power.
sqrt(x)	returns the square root of *x*.

All functions return **double** values, and expect **double** arguments. Since their syntaxes are similar, the following logarithm calculator example suffices for all five of these functions:

```
#include <math.h>
main(argc, argv)
int argc;
char *argv[];
{
   double value;

   sscanf(argv[1], "%lf", &value);
   printf("Natural logarithm of %.12g = %.12g\n", value, log(value));
   printf("Common logarithm of %.12g = %.12g\n", value, log10(value));
}
```

This program accepts its single argument, and returns the natural and common logarithms of that argument.

All five of these functions must be compiled using the **−lm** option to *cc*.

Trigonometric Functions

A full set of trigonometric functions are provided in the math library. They are as follows:

sin(x)	returns the sine of the radian argument *x*.
cos(x)	returns the cosine of the radian argument *x*.
tan(x)	returns the tangent of the radian argument *x*.
asin(x)	returns the arc sine of *x* in the range -pi/2 to pi/2, where $-1 <= x <= 1$.
acos(x)	returns the arc cosine of *x* in the range 0 to pi, where $-1 <= x <= 1$.
atan(x)	returns the arc tangent of *x* in the range -pi/2 to pi/2.
atan2(y, x)	returns the arc tangent of *y*/*x* in the range -pi to pi.
sinh(x)	returns the hyperbolic sine of the radian argument *x*.
cosh(x)	returns the hyperbolic cosine of the radian argument *x*.
tanh(x)	returns the hyperbolic tangent of *x*.

The following program uses some of these routines, as well as two routines from the previous section, to obtain the dimensions and angles of a right triangle:

```
#include <stdio.h>
#include <math.h>
main()
{
   double sideA, sideB, sideC, anga, angb, tempC;
   double pi = fabs(acos(-1.));
   double torads = pi/180.;
   double todegs = 180./pi;
   double angc = 90.;

   printf("Using the following conventions for sides and angles:\n");
   triangle();
   printf("\nEnter all known information:\n");
   printf("\tA = ");
   scanf("%lf", &sideA);
   printf("\tB = ");
   scanf("%lf", &sideB);
   printf("\tC = ");
   scanf("%lf", &sideC);
   printf("\tAngle a = ");
   scanf("%lf", &anga);
   printf("\tAngle b = ");
   scanf("%lf", &angb);
```

```
if(sideA && sideB && sideC) {
     tempC = sqrt(pow(sideA, 2.) + pow(sideB, 2.));
     if(fabs(sideC - tempC) > 0.001) {
          printf("Sides invalid.\n");
          exit(1);
     }
     anga = acos(sideB/sideC) * todegs;
     angb = 90. - anga;
} else if(sideA && sideB) {
     sideC = sqrt(pow(sideA, 2.) + pow(sideB, 2.));
     anga = acos(sideB/sideC) * todegs;
     angb = 90. - anga;
} else if(sideB && sideC) {
     sideA = sqrt(pow(sideC, 2.) - pow(sideB, 2.));
     anga = acos(sideB/sideC) * todegs;
     angb = 90. - anga;
} else if(sideA && sideC) {
     sideB = sqrt(pow(sideC, 2.) - pow(sideA, 2.));
     anga = acos(sideB/sideC) * todegs;
     angb = 90. - anga;
} else if(sideA) {
     if(anga && angb) {
          sideC = sideA/cos(angb*torads);
          sideB = sqrt(pow(sideC, 2.) - pow(sideA, 2.));
     } else if(anga) {
          sideC = sideA/sin(anga*torads);
          sideB = sqrt(pow(sideC, 2.) - pow(sideA, 2.));
          angb = 90. - anga;
     } else if(angb) {
          sideC = sideA/cos(angb*torads);
          sideB = sqrt(pow(sideC, 2.) - pow(sideA, 2.));
          anga = 90. - angb;
     } else {
          printf("Insufficient information.\n");
          exit(1);
     }
} else if(sideB) {
     if(anga && angb) {
          sideC = sideB/sin(angb*torads);
          sideA = sqrt(pow(sideC, 2.) - pow(sideB, 2.));
     } else if(anga) {
          sideC = sideB/cos(anga*torads);
          sideA = sqrt(pow(sideC, 2.) - pow(sideB, 2.));
          angb = 90. - anga;
     } else if(angb) {
          sideC = sideB/sin(angb*torads);
          sideA = sqrt(pow(sideC, 2.) - pow(sideB, 2.));
          anga = 90. - angb;
     } else {
          printf("Insufficient information.\n");
```

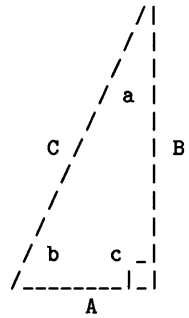
```
            exit(1);
        }
    } else if(sideC) {
        if(anga && angb) {
            sideA = sideC * cos(angb*torads);
            sideB = sideC * sin(angb*torads);
        } else if(anga) {
            sideA = sideC * sin(anga*torads);
            sideB = sideC * cos(anga*torads);
            angb = 90. - anga;
        } else if(angb) {
            sideA = sideC * cos(angb*torads);
            sideB = sideC * sin(angb*torads);
            anga = 90. - angb;
        } else {
            printf("Insufficient information.\n");
            exit(1);
        }
    } else {
        printf("Insufficient information.\n");
        exit(1);
    }

    printf("\n\tSide A = %.2f\t\tAngle a = %.2f degrees\n", sideA, anga);
    printf("\tSide B = %.2f\t\tAngle b = %.2f degrees\n", sideB, angb);
    printf("\tSide C = %.2f\n", sideC);
}
triangle()
{
    FILE *fopen(), *tri;
    char line[50], *fgets();

    tri = fopen("triangle", "r");
    if(tri == NULL) {
        printf("Cannot open triangle file.\n");
        exit(1);
    }

    while(fgets(line, 50, tri) != NULL)
        fputs(line, stdout);
    fclose(tri);
}
```

The *triangle* function prints out the contents of a file in the current directory called **triangle**. The contents of this file should contain an ASCII approximation of a right triangle:



This triangle made up of slashes, vertical bars, and underscores, showing the naming convention for the sides and angles. The program then asks for the known data; enter a value of zero for those parameters that are unknown. The dimensions and angles are then calculated based on the data you have supplied. If there is insufficient information, you are told about it.

The hyperbolic functions are found under *sinh*(3M) in the HP-UX Reference manual. All others are found under *trig*(3M). Thus, the **−lm** argument must be used when compiling code containing these functions.

Miscellaneous Functions

Calculating Upper and Lower Bounds

Two functions, *floor* and *ceil* (see *floor*(3M)), enable you to obtain integers (returned as **doubles**) defining an upper and a lower bound for a number or a series of numbers. *Floor* returns a double precision representation of the the largest integer which is still not greater than *floor*'s argument. Similarly, *ceil* returns a double precision representation of the smallest integer which is still greater than *ceil*'s argument.

The following program returns the floor and ceiling values for the number specified as its argument:

```
#include <math.h>
main(argc, argv)
int argc;
char *argv[];
{
   double value;

   sscanf(argv[1], "%lf", &value);
   printf("Floor = %g; Ceiling = %g\n", floor(value), ceil(value));
}
```

If you type this in and run it, you see that *floor* and *ceil* provide two **double** values representing the smallest range in which the numbers used to obtain that range will fit. For example, if you have a program which reads three values from a source file, and these values are 4.79, 19.6, and 21.1, you can get the smallest possible range in which these numbers fit by running *floor* on each number (and keeping the smallest floor value), and then running *ceil* on each number (and keeping the largest ceiling value). For the above three numbers, this yields a floor value of 4, and a ceiling value of 22.

Code containing these functions must be compiled using the **−lm** *cc* option. **Math.h** need not be included if you remember to explicitly declare that these functions return **double** values.

Calculating Remainders

This section covers two functions, *fmod* and *modf*. The *fmod* function (see *floor*(3M)) returns the remainder (in double precision form) resulting from dividing *fmod*'s first argument by its second. For example,

```
fmod(10., 4.)
```

divides 10 by 4, and returns the remainder (2, in this case). The following program accepts two numbers, divides the first by the second, and displays the results in a form showing the number of times the divisor goes evenly into the dividend, and the remainder, if any.

```
#include <math.h>
main(argc, argv)
int argc;
char *argv[];
{
   int result;
   double number, div, rem;

   sscanf(argv[1], "%lf", &number);
   sscanf(argv[3], "%lf", &div);

   result = number/div;
   printf("%g = (%d)(%g)", number, result, div);
   if((rem = fmod(number, div)) != 0.0)

printf(" + %g\n", rem);
}
```

This program is set up so that it can be invoked in sentence style. If you name the compiled version of this program "divide", then you can say

```
$ divide 33.27 by 11
```

Since argv[2] is ignored in the code, "by" is harmless, and the two numbers are parsed correctly.

Code containing a call to *fmod* must be compiled with the **−lm** *cc* option. However, you need not include **math.h** in your program, as long as you declare *fmod*'s return type appropriately.

The other function, *modf* (see *frexp*(3C)), is not really a remainder function in the same sense that *fmod* is a remainder function. In *fmod*, a division actually takes place. In *modf*, however, no division takes place. *Modf* simply accepts a **double** value, and splits it into its integer and fractional parts. Its syntax is

```
modf(value, iptr);
```

where *value* is the number to be split into two parts, and *iptr* is a pointer to a **double** variable where the integer part of *value* is to be stored. *Modf*'s return value is the signed fractional part of *value*.

The following program shows *modf* in action:

```
main(argc, argv)
int argc;
char *argv[];
{
   double value, iptr, frac, modf();

   sscanf(argv[1], "%lf", &value);
   frac = modf(value, &iptr);
   printf("Integer part: %g; Fractional part: %g\n", iptr, frac);
}
```

The program accepts one argument, the value, and then prints the integer and fractional parts of that value. Note that the address of *iptr* is passed to *modf*, because *modf* expects the address of a **double** variable where the integer part can be stored.

Code containing calls to *modf* does not require the **−lm** option during compilation. Also, the **math.h** include file is of no use to *modf*, so it can be omitted.

Calculating A Hypotenuse

The *hypot* function (see *hypot*(3M)) returns the square root of the sum of the squares of its two arguments, yielding the length of the hypotenuse of a right triangle, or the Euclidian Distance.

Thus, in the previous program which calculated the sides and angles of a right triangle, the line of code which read

```
sideC = sqrt(pow(sideA, 2.) + pow(sideB, 2.));
```

could be replaced with

```
sideC = hypot(sideA, sideB);
```

thus eliminating one function call (*hypot* contains a call to *sqrt*).

Code containing calls to *hypot* must be compiled using the **−lm** option to *cc*.

Generating Random Numbers

The *rand* and *srand* routines (see *rand*(3C)) exist for the generation of random numbers. *Rand* is the random number generator itself, and *srand* enables you to specify a starting point (or *seed*) for *rand*.

The following program simply sets up an infinite loop and lets *rand* run for awhile (to terminate it, just press BREAK, or its equivalent):

```
main()
{
   unsigned value;

   srand(1);
   for(;;) {
       value = rand();
       printf("Random number is %u\n", value);
       sleep(1);
   }
}
```

Note that *rand* and *srand* deal only with *unsigned* integers. If you let this program run for awhile, you'll notice that the random values returned are quite large, and don't often venture below 1000. If your application requires smaller random numbers, divide the value returned by *rand* by some appropriate divisor until a number in the desired range is obtained.

Srand initializes the random number generator to a particular starting point. In the above program, 1 is used, but you can specify any positive integer you like.

The *sleep* library routine causes the program to "pause" for the number of seconds specified (1, in this case).

Floating-Point Exponentiation Routines

Two routines, *frexp* and *ldexp* (see *frexp*(3C)), are covered in this section. *Frexp* accepts a **double** value, and returns two values, *x* and *n*, such that

```
value = x * 2^n
```

where *x* is a **double** quantity of magnitude less than 1, and *n* is an integer exponent. *Frexp*'s syntax is

```
frexp(value, eptr);
```

where *value* is the value to be processed, and *eptr* is a pointer to an integer variable where the exponent *n* is to be stored. The quantity *x* is returned as *frexp*'s return value.

The following program accepts a number argument and uses *frexp* to output that number's representation in the form shown above:

```
main(argc, argv)
int argc;
char *argv[];
{
   double value, x, frexp();
   int eptr;

   sscanf(argv[1], "%lf", &value);
   x = frexp(value, &eptr);
   printf("%g = %g * 2^%d\n", value, x, eptr);
}
```

Ldexp accepts a **double** *value* and an integer exponent *exp*, and returns a **double** quantity equal to

```
<value> * 2^ <exponent>
```

The following program accepts two number arguments, *value* and *exp*, and outputs the result:

```
main(argc, argv)
int argc;
char *argv[];
{
   double value, result, ldexp();
   int exp;

   sscanf(argv[1], "%lf", &value);
   sscanf(argv[2], "%d", &exp);
   result = ldexp(value, exp);
   printf("%g * 2^%d = %g\n", value, exp, result);
}
```

Neither of these routines require **math.h** or the use of the **−lm** *cc* option.

Part 3: String Manipulations

Character Conversion and Classification

This section discusses those routines found under *conv*(3C) and *ctype*(3C) which enable you to convert between upper- and lower-case, and classify characters as digits, non-printing, upper-case, etc.

Converting Between Uppercase and Lowercase

Four routines are documented under *conv*(3C) which enable you to convert between upper- and lowercase. They are *toupper*, *tolower*, *_toupper*, and *_tolower*.

Toupper and *tolower* are functions which accept a single integer argument in the range −1 through 255. If the integer taken as a character represents a lower-case character, *toupper* returns the corresponding upper-case character. Similarly, *tolower* returns the corresponding lower-case character. Both routines return the argument unchanged if it does not represent a lower-case character (*toupper*) or an upper-case character (*tolower*).

_toupper and *_tolower* are macros defined in **ctype.h**. *_toupper* accepts a single integer argument which *must* represent a lower-case character; the corresponding upper-case character is returned. Similarly, *_tolower must* be given an upper-case character, and returns the corresponding lower-case character. If an argument is specified which is not a lower-case character (*_toupper*) or an upper-case character (*_tolower*), garbage is returned.

The macro versions of these routines are faster than the functions, so if you can guarantee that only lower-case or upper-case characters are passed to the macros, you should probably use them. However, the function versions are handy for tasks like

```
...
for(i=0; array[i] != NULL; i++)
     array[i] = toupper(array[i]);
...
```

which converts every lowercase character found in *array* to uppercase. The functions enable you to be more lenient about the arguments passed to them. In the above program fragment, no argument checking is needed; if the argument isn't a lowercase character, it is returned unchanged.

Character Classification

The *ctype*(3C) entry in the HP-UX Reference lists routines which test their single argument and return a non-zero value if the test is positive, and 0 otherwise.

All of these routines are macros defined in **ctype.h**. Because their syntaxes are identical, the following example suffices for all *ctype* macros:

```
...
for(i=0; array[i] != NULL; i++) {
     if(islower(array[i]))
          array[i] = _toupper(array[i]);
}
...
```

This program fragment shows one way to change all occurrences of a lower-case character in *array* to upper-case using the macro *_toupper*. The macro *islower* is used to make sure that only lower-case characters are passed to *_toupper*.

String Manipulation

String(3C) in the HP-UX Reference manual documents an extensive list of string manipulation routines enabling you to perform several operations on character strings. This section describes the *string*(3C) package in detail.

Concatenating Strings

Strcat and *strncat* enable you to append a copy of one string onto the end of another. Their syntaxes are:

```
strcat(s1, s2);
strncat(s1, s2, n);
```

where *s1* and *s2* are character pointers to NULL-terminated character strings. *Strcat* appends the entire string pointed to by *s2* (up to the first NULL character encountered) onto the end of string *s1*. *Strncat* does the same thing, except that at most *n* characters are appended to *s1* (or up to a NULL character, whichever comes first). (Note that string *s2* need not be NULL-terminated when using *strncat* if *n* is less than or equal to the length of *s2*.) Both routines return a character pointer to the NULL-terminated result.

Neither of these routines checks to make sure that there is room in *s1* for the additional characters of *s2*. Thus, to be safe, *s1* should **always** be a declared array having plenty of space for the additional characters of *s2*, plus a terminating NULL character.

Copying Strings

Strcpy and *strncpy* copy one string of characters into another. Their syntaxes are:

```
strcpy(s1, s2);
strncpy(s1, s2, n);
```

where *s2* is a character pointer to the string to be copied, and *s1* is a character pointer to the beginning of the string into which the contents of string *s1* are copied. *Strcpy* copies the entire string, up to (and including) the first NULL encountered. *Strncpy* copies up to *n* characters, or up to (and including) the first encountered NULL, whichever occurs first. (String *s2* need not be NULL-terminated when using *strncpy* if *n* is less than or equal to the length of *s2*.) Both routines return the value of *s1*.

The following program uses the *strcat* routine discussed earlier and *strcpy* to build a character string representing the lower-case alphabet, one character at a time.

```
#include <stdio.h>
main()
{
   int b = 'b', z = 'z', i;
   char alpha[30], chr[4];

   chr[1] = NULL;
   strcpy(alpha, "a");
   printf("%s\n", alpha);

   for(i = b; i <= z; i++) {
        chr[0] = i;
        strcat(alpha, chr);
        printf("%s\n", alpha);
   }
}
```

The array *chr* is always going to be a two-character array consisting of the next character in the alphabet followed by NULL. Thus, the second element of *chr* is set to NULL early in the program. The first *chr* element is then successively set to the next lower-case character in the **for** loop, and the resulting two-character string is concatenated onto the end of the alphabet assembled so far in *alpha*. Note the use of *strcpy* to initialize *alpha*. Remember that C transforms one or more characters enclosed in double quotes into a character pointer to those characters followed by a NULL. Thus, the *strcpy* statement above copies the character "a" followed by a NULL character into *alpha*.

There are some things to be aware of when using *strcat*, *strncat*, *strcpy*, and *strncpy*. These routines all modify string *s1* in some way, but none of them check for overflow in that string. Therefore, be sure there is enough room in *s1* to hold the added or copied characters plus a terminating NULL. Also, be sure you use a character *array* for *s1* (*not* just a character pointer), especially when using *strcat* or *strncat*. This is because an explicitly-declared array has sufficient memory allocated to it to contain all of its elements, but a character pointer simply points to a single location in memory. Concatenating a string to the end of a string contained in an array is guaranteed to work, provided the array is large enough. However, concatenating a string to a string of characters referenced by a simple character pointer is dangerous, since the concatenated characters could overwrite data in memory. For example,

```
char array[100], *ptr = "abcdef";
...
strcat(array, ptr);
```

works fine, since you are guaranteed that 100 storage elements have been set aside for the array. However,

```
char *ptr1 = "abcdef", *ptr2 = "ghijkl";
...
strcat(ptr1, ptr2);
```

is asking for trouble. Although C makes sure that there is enough room for the initializing strings ("abcdef" and "ghijkl" in this example), there are no guarantees that there is enough room to add characters to the end of one of these strings. Therefore, the last fragment could easily overwrite valid data occurring after the string pointed to by *ptr1*.

Since string *s2* is not modified, you can use arrays or character pointers with no ill effects.

Comparing Strings

Strcmp and *strncmp* compare two strings and return an integer indicating the result of the comparison. Their syntaxes are:

```
strcmp(s1, s2);
strncmp(s1, s2, n);
```

where *s1* and *s2* are character pointers to the NULL-terminated character strings to be compared. *Strcmp* compares the entire strings, stopping as soon as the result is determined. *Strncmp* compares at most *n* characters of both strings (neither string need be NULL-terminated if *n* is less than or equal to the length of the shorter string). The integer returned uses the following convention:

<0	*s1* is lexicographically less than *s2*;
=0	*s1* and *s2* are equal;
>0	*s1* is lexicographically greater than *s2*.

The following program fragment uses *strncmp* to analyze the contents of a file coded with the man macros (see *man*(7)). It reads each line of the file and keeps a count of the number of times selected macros are used, and prints a summary of its findings at the end.

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
   char *fgets(), line[100];
   FILE *fp;
   int nsh, npp, ntp, nrs, nre, npd, nip, nmisc, nlines;
```

```
    nsh = npp = ntp = nrs = nre = npd = nip = nmisc = nlines = 0;

    if(argc != 2) {
         fprintf(stderr, "Usage: count file\n");
         exit(2);
    }

    fp = fopen(argv[1], "r");
    if(fp == NULL) {
         fprintf(stderr, "Can't open %s.\n", argv[1]);
         exit(1);
    }

    while(fgets(line, 100, fp) != NULL) {
         if(strncmp(line, ".SH", 3) == 0)
              nsh++;
         else if(strncmp(line, ".PP", 3) == 0)
              npp++;
         else if(strncmp(line, ".TP", 3) == 0)
              ntp++;
         else if(strncmp(line, ".RS", 3) == 0)
              nrs++;
         else if(strncmp(line, ".RE", 3) == 0)
              nre++;
         else if(strncmp(line, ".PD", 3) == 0)
              npd++;
         else if(strncmp(line, ".IP", 3) == 0)
              nip++;
         else if(line[0] == '.')
              nmisc++;
         nlines++;
    }

    printf("No. of lines: %d\n\n", nlines);
    printf("No. of .SH's: %d\n", nsh);
    printf("No. of .PP's: %d\n", npp);
    printf("No. of .TP's: %d\n", ntp);
    printf("No. of .RS's: %d\n", nrs);
    printf("No. of .RE's: %d\n", nre);
    printf("No. of .PD's: %d\n", npd);
    printf("No. of .IP's: %d\n", nip);
    printf("No. of misc. macros: %d\n", nmisc);

    fclose(fp);
    exit(0);
}
```

In the above program, *strncmp* is used to compare the first three characters of each line read. If the first three characters match a particular macro, the appropriate counter is incremented. If the line begins with ".", but is not one of the macros being searched for, the "miscellaneous" counter is incremented. The total number of lines in the file is also given.

Finding the Length of a String

The *strlen* routine returns an integer specifying the number of non-NULL characters in a string. Its syntax is:

```
strlen(s);
```

where *s* is a character pointer to the NULL-terminated string whose length is to be taken. For example, if you execute

```
len = strlen(string);
```

then the integer *len* contains the total number of non-\s-1NULL\s+1 characters in the string pointed to by *string*. Thus,

```
string[len]
```

points to the terminating NULL in *string*

Finding Characters in Strings

The *strchr*, *strrchr*, and *strpbrk* routines enable you to locate a particular character within a string.

Strchr and *strrchr* return a character pointer to an occurrence of a specified character in a string. Their syntaxes are:

```
strchr(s, c);
strrchr(s, c);
```

where *s* is a character pointer to the string of interest, and *c* is a variable of type **char** specifying the character to search for.

Strchr returns a character pointer to the first occurrence of character *c* in string *s*. Similarly, *strrchr* returns a character pointer to the last occurrence in string *s*. Both routines return a NULL if the character does not occur in the string pointed to by *s*. For example,

```
char *ptr, *strchr(), string[100];
...
while((ptr = strchr(string, '@') != NULL)
   *ptr = '#';
```

replaces all occurrences of "@" in the array *string* with "#", starting from the beginning of the array and working toward the end. The same operation can be done using

```
while((ptr = strrchr(string, '@')) != NULL)
   *ptr = '#';
```

which replaces all @'s with #'s, starting from the end of the array, working backward toward the beginning.

The *strpbrk* routine returns a character pointer to the first occurrence in string *s1* of any character contained in string *s2*, or NULL if none of the characters in *s2* occur in *s1*. Its syntax is:

```
strpbrk(s1, s2);
```

For example, suppose you have to read lines of input in which are embedded numerical data which must be read. For simplicity, assume that the following conventions are used:

- Positive numbers do not begin with "+";
- Fractional numbers always begin with zero, as in 0.25;
- The first occurrence of a digit in the string signals the beginning of the number to be read.

Given these rules, the following code fragment does the job:

```
char line[100], *chrs = "-0123456789", *ptr;
float value;
...
ptr = strpbrk(line, chrs);
sscanf(ptr, "%f", &value);
...
```

The character pointer *chrs* is initialized to point to a string of characters which might introduce the embedded number. *Strpbrk* then finds the first occurrence of one of these characters in *line*, and returns a pointer to that location in *ptr*. Finally, *ptr* is passed to *sscanf*, which interprets *ptr* as if it were a pointer to the beginning of a string from which input is to be taken. The number is read correctly because *ptr* points to the beginning of a number, and because the %f conversion terminates at the first inappropriate character.

Miscellaneous String Routines

Finding Characters Common to Two Strings

The *strspn* and *strcspn* routines return an integer giving the length of the initial segment of string *s1* which consists entirely of characters found in string *s2*. *Strcspn* is similar, but returns an integer giving the length of the initial segment of *s1* which consists entirely of characters *not* found in string *s2*. Their syntaxes are:

```
strspn(s1, s2);
strcspn(s1, s2);
```

For example, suppose you have the following two strings:

```
"A tattle-tale never wins."
```

for string *s1*, and

```
" -Aatle"
```

for *s2*. Executing

```
strspn(s1, s2);
```

with the strings shown returns a value of 14, since the first 14 characters in *s1* all occur in *s2* — "A tattle-tale ". If you execute

```
strcspn(s1, s2);
```

using the same strings, you get 0, because there is no initial segment of *s1* which contains characters not found in *s2*.

Breaking a String into Tokens

A *token* is a string of characters delimited by one or more token delimiters. The *strtok* routine divides string *s1* into one or more tokens. The token separators consist of any characters contained in string *s2*. Its syntax is:

```
strtok(s1, s2);
```

where *s1* is a character pointer to the string which is to be broken up into tokens, and *s2* is a character pointer to a string consisting of those characters which are to be treated as token separators.

Strtok returns the next token from *s1* each time it is called. The first time *strtok* is called, both *s1* and *s2* must be specified. On subsequent calls, however, *s1* need not be specified (a NULL is specified in its place). *Strtok* remembers the string from call to call. String *s2* must be specified each call, but need not contain the same characters (token separators) each time.

Strtok returns a pointer to the beginning of the next token, and writes a NULL character into *s1* immediately following the end of the returned token. *Strtok* returns a NULL when no tokens remain.

For example, suppose you are reading lines from */etc/gettydefs*, which is the speed table for *getty*(1M) – see *gettydefs*(5). The lines in this file contain several fields delimited by pound signs (#). Thus, the following code could be used to read the fields of each line:

```
int count = 0;
char *delims = "#", *token, *arg1, *strtok(), line[256];
arg1 = line;
...
while((token = strtok(arg1, delims) != NULL) {
   count++;
   printf("field %d: %s\n", count, token);
   if(count == 1)
      arg1 = NULL;
}
```

This code sees to it that *strtok*'s first argument is NULL after the first call. Also, note that *delims* did not change from call to call, but it could have. This greatly increases the power of *strtok*, since it enables you to change the token delimiters between calls.

Part 4:
Date and Time Manipulation

Ctime(3C) describes a set of routines which enable you to access the date and time as maintained by the system clock. This package knows about daylight saving time, and automatically converts between standard time and daylight saving time when appropriate.

Most of the *ctime* routines require the quantity returned by *time*(2), which is the number of seconds that have elapsed since 00:00:00 GMT (Greenwich Mean Time), January 1, 1970.

The *ctime* routine converts the *time*(2) value into a 26-character ASCII string of the form

```
Fri May 11 09:53:03 1984\n\0
```

where "\n" is a new-line character, and "\0" is a terminating NULL character. *Ctime*'s syntax is:

```
ctime(value);
```

where *value* is a pointer to a long integer value representing the number of elapsed seconds since 00:00:00 GMT, January 1, 1970 (as returned by *time*(2)). Note that *value* is a pointer to the quantity returned by *time*(2), not just the quantity itself. Using *time*(2) and *ctime*, you can write your own simplified version of the *date*(1) command:

```
#include <stdio.h>
main()
{
   char *str, *ctime();
   long time(), nseconds;

   nseconds = time((long *)0);
   str = ctime(&nseconds);
   printf("%s", str);
}
```

The rest of the routines in *ctime*(3C) require the include file *time.h*, which contains the definition of a structure called *tm*. This structure is made up of several variables which contain the various components of the date and time. It looks as follows:

```
struct tm {
     int  tm_sec;
     int  tm_min;
     int  tm_hour;
     int  tm_mday;
     int  tm_mon;
     int  tm_year;
     int  tm_wday;
     int  tm_yday;
     int  tm_isdst;
};
```

The meaning associated with each structure member is:

tm_sec	the "seconds" portion of the system's 24-hour clock time;
tm_min	the "minutes" portion of the system's 24-hour clock time;
tm_hour	the "hours" portion of the system's 24-hour clock time;
tm_mday	the day of the month, in the range 1 thru 31;
tm_mon	the month of the year, in the range 0 thru 11 (0 = January);
tm_year	the current year − 1900;
tm_wday	the day of the week, in the range 0 thru 6 (0 = Sunday);
tm_yday	the day of the year, in the range 0 thru 365;
tm_isdst	a flag which is non-zero if daylight saving time is in effect.

The *localtime* and *gmtime* routines accept a pointer to a quantity such as returned by *time*(2), and fill in the various components of the *tm* structure. *Localtime* corrects the time for the local time zone and possible daylight saving time, while *gmtime* converts directly to GMT time (this is the time used by HP-UX). Both routines return a pointer to a structure of type *tm* which can be used to access the various components of the *tm* structure.

For example, the following code fragment assigns values to the *tm* structure members for the local time zone:

```
#include <time.h>
...
struct tm *ptr, *localtime();
long time(), nseconds;
...
nseconds = time((long *)0);
ptr = localtime(&nseconds);
```

Once this code is executed, you can use *ptr* to access the different components of the local time. For example, **ptr->tm_mon** references the month of the year, and **ptr->tm_wday** references the day of the week. (*Gmtime* is used in exactly the same way, so this example suffices for it also).

The *asctime* routine converts the time contained in a *tm* structure into \s-1ASCII\s+1 representation such as that returned by *date*(1) and *ctime*. Its syntax is:

```
asctime(ptr);
```

where *ptr* is a pointer to a structure of type *tm* whose members have previously been assigned values with *localtime* or *gmtime*, or explicitly by you. *Asctime* returns a character pointer to the same NULL-terminated 26-character string as returned by *ctime*.

Asctime provides a way for you to obtain the current time, modify it explicitly in some way, and then print the result in ASCII form. The *date* command shown earlier can be re-written using *localtime* and *asctime*:

```
#include <stdio.h>
#include <time.h>
main()
{
   long time(), nseconds;
   struct tm *ptr, *localtime();
   char *string, *asctime();

   nseconds = time((long *)0);
   ptr = localtime(&nseconds);

/* the user may modify the current time in tm here */

   string = asctime(ptr);
   printf("%s", string);
}
```

This program illustrates a rather indirect way to obtain the date, but it does enable you to modify the date stored in *tm* before you print it out. If all you want to do is print the date, the quickest way is to use the *time/ctime* combination.

Of all the *ctime* routines, perhaps the most useful is *localtime*. It enables you to break the current time up into referencable chunks which can then be examined for such applications as personal calendar programs, program schedulers, etc. Many of the *tm* values can be used as indices into arrays containing strings identifying months and days. For example, declaring an external array like

```
char *month[] = { "January", "February", "March", "April",
                  "May", "June", "July", "August", "September",
                  "October", "November", "December"
                };
```

enables you to use **tm_mon** as an index into this array to obtain the actual month name. The same thing can be done with **tm_wday** if you initialize an array containing the names of the days of the week. The *ctime*(3C) package makes it easy to design programs which depend upon the time or date. Try creating your own versions of *calendar*(1), *at*(1), or even *cron*(1M)!

Index

a

abs (absolute value function) 62
acos function 64
asctime function 85
asin function 64
atan function 64
atan2 function 64

b

breaking strings into tokens 82
BUFSIZ 3

c

calculating remainders 69
char declaration replaced by int 5
character classification 74
character conversion characters 9, 17
character file I/O 29–32
character push-back 32
characters, character conversion 9, 17
characters, floating-point conversion 10, 17
characters, format conversion 8, 16
characters, integer conversion 9, 16
characters, literal 11, 14
clearerr (clear file read/write error status) 44
comparing strings 77
concatenating strings 75
conversion characters, character 9, 17
conversion characters, floating-point 10, 17
conversion characters, format 8, 16
conversion characters, integer 9, 16
conversion specifications, format 7
converting between file pointers and file descriptors 55
copying strings 75
cos function 64

cosh function 64
ctime function 83
ctype function 74

d

date and time manipulation 83–86

e

exponentiation function 63
exponentiation function, floating-point 72

f

fabs (floating-point absolute value) 62
fclose 50
fdopen 56
feof (end-of-file status inquiry) 42
ferror (read/write error file status inquiry) 44
fflush 53
fflush before converting file pointer/file descriptor 55
fgetc (get character from a file) 31
fgets 33
file descriptor/file pointer conversion 55
file I/O:
 binary (non-ASCII) data 36
 formatted 35–36
 single-character 29–32
 strings 33
file open for read and write 49
file pointer 27
file pointer/file descriptor conversion 55
fileno 56
filters 6
finding characters common to two strings 81
floating-point exponentiation function 72
fmod function 69
fopen 27–28
format conversion characters 8, 16
format conversion specifications 7, 15
formatted I/O 7
fprintf 35

fputc (put character in a file) 31
fputs 33
fread 38–41
freopen 54
frexp function 71
fscanf 35
fseek 46
ftell 46
fwrite 38–41

g

gets function reads strings into array 6
getw 36
gmtime function 85

h

hypotenuse 70

i

int declared instead of char 5
integer conversion characters 9, 16
inter-process communication 58–60
I/O:
 formatted 7
 ordinary files 26
 single-character 5
 string 6
 strings 21–25

l

ldexp function 71
literal characters 11, 14
localtime function 85
logarithmic functions 63
lowercase/uppercase conversion 73
lower/upper bounds for numbers 68

m

math routines 61–72
math.h include file 61
modf function 69, 70

o

offset from file/stream pointer 46
open ordinary file 26

p

popen 58
power math function 63
printf examples 18–20
printf for formatted output 7
printf output formatting 14–20
putchar to output newline 5
puts function copies character array to stdout 6
putw 36

r

rand function 71
random numbers 71
remainders, calculating 69
reposition stream (file) I/O operations 45, 49
rewind 45

s

scanf for formatted input 7
setbuf 50
setvbuf 52
sin function 64
single-character I/O 5
sinh function 64
specifications, format conversion 7
square root function 63
srand function 71
standard error (see stderr) 1–4
standard input (see stdin) 1–6
standard output (see stdout) 1–6
stderr 1–6

stdin 1–6
stdout 1–6
strcat function 75
strchr function 77
strcmp function 77
strcspn function 81
stream (file pointer) control routines 50–55
stream status 42
string file I/O 33
string I/O 6, 21–25
string manipulations 73–82
string:
 read data from 21
 write data to 24
strings:
 breaking into tokens 82
 comparing 77
 concatenating 75
 copying 75
 finding characters common to two strings 81
 finding characters in 77
 finding length of 77
strlen function 77
strncat function 75
strncmp function 77
strrchr function 77
strspn function 81
strtok function 82

t

tan function 64
tanh function 64
time and date manipulation 83–86
_tolower 73–74
_toupper 73–74
triangle function 67
trigonometric functions 64

u

ungetc (place character back on input stream) 32
uppercase/lowercase conversion 73
upper/lower bounds for numbers 68

Table of Contents

Lint C Program Checker

Introduction 1
Error Detection 1
Problem Detection 2
Problem Code: Unused Variables and Functions 3
Problem Code: Set/Used Information 4
Problem Code: Unreachable Code 5
Problem Code: Function Value 5
Problem Code: Type Matching 6
Problem Code: Portability 8
Problem Code: Strange Constructions 10
How to Use Lint 12
Directives 14

Lint C Program Checker

Introduction

Lint is a program checker and verifier for C source code. Its main purpose is to supply the programmer with warning messages about problems with the source code's style, efficiency, portability, and consistency. Once the C code passes through the compiler with no errors, *lint* can be used to locate areas undetected by the compiler that may require corrections.

Error messages and *lint* warnings are sent to the standard error file (the terminal by default). Once the code errors are corrected, the C source file(s) should be run through the C compiler to produce the necessary object code.

Error Detection

Lint detects all code errors that can be detected by the C compiler and produces an error message such as:

```
illegal initialization
```

These errors must be corrected before the compiler can be used to produce object code.

While *lint* can be used for error detection, it cannot recover from all of the code errors it finds. If *lint* encounters an error that it cannot recover from, it terminates after sending the message:

```
cannot recover from earlier errors--goodbye!
```

If *Lint* detects more than its limit of 30 coding errors in the source file(s), it terminates after sending the error message:

```
too many errors
```

Since *lint* cannot recover from certain errors, and because of the limited number of errors it can handle, the compiler should be used instead for error detection. After all errors that the compiler can detect have been corrected, *lint* can be used to help find bugs and inefficiencies in the source code.

Problem Detection

Remember that a compiler reports errors only when it encounters program source code that cannot be converted into object code. The main purpose of *lint* is to find problem areas in C source code that it considers to be inefficient, nonportable, bad style, or a possible bug, but which the C compiler accepts as error-free because it can be converted into object code.

Comments about problems that are local to a function are produced as each problem is detected. They have the form:

```
warning: <message text>
```

Information about external functions and variables is collected and analyzed after *lint* has processed the files handed to it. At that time, if a problem has been detected, it sends a warning message with the form:

```
<message text>
```

followed by a list of external names causing the message and the file where the problem occurred.

Code causing *lint* to issue a warning message should be analyzed to determine the source of the problem. Sometimes the programmer has a valid reason for writing the problem code. Usually, though, this is not the case. *Lint* can be very helpful in uncovering subtle programming errors.

Lint checks the source code for certain conditions, about which it issues warning messages. These can be grouped into the following categories:

1. variable or function is declared but not used;
2. variable is used before it is set;
3. portion of code is unreachable;
4. function values are used incorrectly;
5. type matching does not adhere strictly to C rules;
6. code has portability problems;
7. code construction is strange.

The code that you write may have constructions in it that *lint* objects to but that are necessary to its application. Warning messages about problem areas that you know about and do not plan to correct provide useless information and make helpful messages harder to find. There are two methods for suppressing warning messages from *lint* that you do not need to see. The use of *lint options* is one. The *lint* command can be called with any combination of its defined option set. Each option has *lint* ignore a different problem area. Ths other method is to insert *lint directives* into the source code. *Lint* directives are discussed later.

Problem Code: Unused Variables and Functions

Lint objects if source code declares a variable that is never used or defines a function that is never called. Unused variables and functions are considered bad style because their declarations clutter the code. They can also be the cause of a program bug if their use is essential.

An unused local variable can result in one of two *lint* warning messages. If a variable is defined to be static and is not used, *lint* responds with:

```
warning: static variable <name> unused
```

Unused automatic variables cause the message:

```
warning: <name> unused in function <name>
```

A function or external variable that is unused causes the message:

```
name defined but never used
```

followed by the function or variable name and the file in which it was defined. *Lint* also looks at the special case where one of the parameters of a function is not used. The warning message is:

```
warning: argument unused in function: <arg_name> in <func_name>
```

If functions or external variables are declared but never used or defined, *lint* responds with

```
name declared but never used or defined
```

followed by a list of variable and function names and the names of files where they were declared.

Suppressing Lint

Sometimes it is necessary to have unused function parameters to support consistent interfaces between functions. The **-v** option can be used with *lint* to have warnings about unused parameters suppressed. However, the **-v** option does not suppress comments when parameters are defined as register variables. Unused register variables result in inefficient use of computer's resources, because quick-access hardware is frequently allocated for their storage.

If *lint* is run on a file which is linked with other files at compile time, many external variables and functions can be defined but not used, as well as used but not defined. If there is no guarantee that the definition of an external object is always seen before the object code is used, it is declared **extern**. The **-u** option can be used to stop complaints about all external objects, whether or not they are declared **extern**. If you want to inhibit complaints about only the **extern** declared functions and variables, use the **-x** option.

Problem Code: Set/Used Information

A problem bug exists in a program if a variable's value is used before it is assigned. Although *lint* attempts to detect occurrences of this, it takes into account only the physical location of the code. If code using a static or external variable is located before the variable is given a value, the message set is:

```
warning: <name> may be used before set
```

Since static and external variables are always initialized to zero, this may not point out a program bug. *Lint* also objects if automatic variables are set in a function but not used. The message given is:

```
warning: <name> set but not used in function
```

Problem Code: Unreachable Code

Lint checks for three types of unreachable code. Any statement following a **goto, break, continue**, or **return** statement must either be labeled or reside in an outer block for *lint* to consider it reachable. If neither is the case, *lint* responds with:

```
warning:  statement not reached
```

The same message is given if *lint* finds an infinite loop. It only checks for the infinite loop cases of **while(1)** and **for(;;)**. The third item that *lint* looks for is a loop that cannot be entered from the top. If one is found, then the message sent is:

```
warning:  loop not entered from top
```

Lint's detection of unreachable code is by no means perfect. Warning messages can be sent about valid code. It can also overlook commenting on code that canot be reached. An example of this is the fact that *lint* does not know if a called function ever returns to the calling function (e.g., **exit**). *Lint* does not identify code following such a function as being unreachable.

Suppressing Lint

Programs that are generated by *yacc* or *lex* can have many unreachable **break** statements. Normally, each one causes a complaint from *lint*. The **-b** option can be used to force *lint* to ignore unreachable **break** statements.

Problem Code: Function Value

The C compiler allows a function containing both the statement

```
return();
```

and the statement

```
return(expression);
```

to pass through without complaint. *Lint*, however, detects this inconsistency and responds with the message:

```
warning:  function <name> has return(e);and return;
```

The most serious difficulty with this is detecting when a function return is implied by flow of control reaching the end of the function. This can be seen with a simple example:

```
f(a)
{
    if (a) return (3);
    g();
}
```

Notice that if *a* tests false, *f* will call *g* and then return with no defined value. This will trigger a message for *lint*. If *g* (like **exit**) never returns, the message will still be produced when in fact nothing is wrong. In practice, some potentially serious bugs have been discovered by this feature.

On a global scale, *lint* detects cases where a function returns a value that is sometimes or never used. When the value is never used, it may constitute an inefficiency in the function definition. When the value is sometimes used, it may represent bad style (e.g., not testing for error conditions).

The dual problem — using a function value when the function does not return one — is also detected. This is a serious problem.

Problem Code: Type Matching

The C compiler does not strictly enforce the C language type matching rules. At the loss of some type checking, the C compiler gains speed. An important role of *lint* is to enforce the type checking that the compiler neglects. It does this in four areas:

1. pointer types;
2. **long** and **int** type matching;
3. enumerations;
4. operations on structures and unions.

The types of pointers used in assignment, conditional, relational, and initialization statements must agree exactly. For example, the code:

```
int *p
char *q
.
.
.
p = q;
```

would cause *lint* to respond with the message

```
warning:  illegal pointer combination
```

Adding and subtracting integers and pointers are legal. Any other binary operation on them results in the message

```
warning:  illegal combination of pointer and integer:  op <operator>
```

An example of code causing this message would be:

```
int s, *t;
.
.
.
t = s;
```

Assignments of long integer variables are possible in the C language. However, on some machines the amount of storage supplied for the two types differs, and so the accuracy of a value could be lost in the conversion. *Lint* detects these assignments as possible program bugs. If a long integer is assigned to an integer, *lint* responds with:

```
warning:  conversion from long may lose accuracy
```

Lint checks enumerations to see that variables or members are all of one type. Also, the only enumeration operations it allows are are assignment, initialization, equality, and inequality. If *lint* finds code breaking any of these guidelines, it sends the message:

```
warning:  enumeration type clash, operator <operator>
```

Structure and union references are subject to more type checking by *lint* than by the C compiler. *Lint* requires that the left operand of −> be a pointer to a structure or a union. If it isn't a pointer, *lint*'s response is:

```
warning: struct/union or struct/union pointer required
```

The left operand of . must be a structure or a union, which *lint* also indicates with the message above. The right operand of −> and . must be a member of the structure or union implied by the left operand. If it isn't, then *lint*'s message is:

```
warning: illegal member use <name>
```

where <name> is the right operand.

Suppressing Lint

You may have a legitimate reason for converting a long integer to an integer. *Lint*'s **−a** option inhibits comments about these conversions.

Problem Code: Portability

The **−p** option of *lint* aids the programmer in writing portable code in five areas:

1. character comparisons;
2. pointer alignments;
3. uninitialized external variables;
4. length of external variables;
5. type casting.

Character representation varies on different machines. Characters may be implemented as signed values. As a result, certain comparisons with characters give different results on different machines. The expression

```
c<0
```

where *c* is defined as type character, is always true if characters are unsigned values. If, however, characters are signed values, the expression could be either true or false. Where character comparisons could result in different values depending on the machine used, *lint* outputs the message:

```
warning: nonportable character comparison
```

Legal pointer assignments are determined by the alignment restrictions of the particular machine used. For example, one machine may allow double-precision values to begin on any integer boundary, but another may restrict them to word boundaries. If integer and word boundaries are different, code containing an assignment of a double pointer to an integer pointer could cause problems. *Lint* attempts to detect where the effect of pointer assignments is machine dependent. The warning that it sends is:

```
warning:  possible pointer alignment problem
```

Another machine-dependent area is the treatment of uninitialized external variables. If two files both contain the declaration

```
int a;
```

either one word of storage is allocated or each occurrence receives its own word of storage, depending on the machine. If the files that *lint* is processing contain multiple definitions of the same uninitialized external variable, *lint* responds with:

```
warning: <name> redefinition hides earlier one
```

The amount of information about external symbols that is loaded depends on the machine being used: the number of characters saved and whether or not uppercase/lowercase distinction is kept. *Lint* truncates all external symbols to six characters and allows only one case distinction. (It changes uppercase characters to lowercase.) This provides a worst-case analysis so that the uniqueness of an external symbol is not machine-dependent.

The effectiveness of type casting in C programs can depend on the machine that is used. For this reason, *lint* ignores type casting code. All assignments that use it are subject to *lint*'s type checking (see *Problem Code: Type Matching*).

Suppressing Lint

The **-p** option stops comments about two types of portability problems:

1. pointer alignment problems,
2. multiple definitions of external variables.

Lint's objections to legal casts can also be suppressed by using the **-c** option.

Problem Code: Strange Constructions

A *strange construction* is code that *lint* considers to be bad style or a possible bug.

Lint looks for code that has no effect. An example is:

```
*p++;
```

where the * has no effect. The statement is equivalent to `"p++;"`. In cases like this, the message:

```
warning:  null effect
```

is sent.

The treatment of unsigned numbers as signed numbers in comparison causes *lint* to report:

```
warning:  degenerate unsigned comparison
```

The following code would produce such a message:

```
unsigned x;
.
.
.
if (x,0) ...
```

Lint also objects if constants are treated as variables. If the boolean expression in a conditional has a set value due to constants, such as

```
if(1!=0) ...
```

lint's response is:

```
warning:  constant in conditional context
```

If the NOT operator is used on a constant value, the response is:

```
warning:  constant argument to NOT
```

To avoid operator precedence confusion, *lint* encourages using parentheses in expressions by sending the message:

```
warning: precedence confusion possible: parenthesize!
```

Lint judges it bad style to redefine an outer block variable in an inner block. Variables with different functions should normally have different names. If variables are redefined, the message sent is:

```
warning: <name> redefinition hides earlier one
```

Suppressing Lint

To stop *lint*'s comments about strange constructions, use its **−h** option.

How to Use Lint

The *lint* command has the form:

```
lint [options] files ... library-descriptors ...
```

where *options* are optional flags to control *lint* checkng and messages, *files* are the files to be checked which end with **.c** or **.ln**, and *library-descriptors* are the names of libraries to be used in checking the program.

The options that are currently supported by the *lint* command are:

−a	Suppress messages about assignments of long values to variables that are not long.
−b	Suppress messages about break statements that cannot be reached.
−c	Only check for intrafile bugs; leave external information in files suffixed with **.ln**.
−h	Do not apply heuristics (which attempt to detect bugs, improve style, and reduce waste).
−n	Do not check for compatibility with either the standard or the portable *lint* library.
−o *name*	Create a *lint* library from input files named **llib−l***name***.ln**.
−p	Attempt to check portability to other dialects of C language.
−u	Suppress messages about function and external variables used and not defined or defined and not used.
−v	Suppress messages about unused arguments and functions.
−x	Do not report variables referred to by external declaractions but never used.

The names of files that contain C language programs should end with the suffix **.c**, which is mandatory for *lint* and the C compiler.

The *lint* program accepts certain arguments, such as:

```
-ly
```

These arguments specify libraries that contain functions used in the C language program. The source code is tested for compatibility with these libraries. This is done by accessing library description files whose names are constructed from the library arguments. These files all begin with the comment:

```
/*LINTLIBRARY*/
```

which is followed by a series of dummy function definitions. The critical parts of these definitions are the declaration of the function return type, whether the dummy function returns a value, and the number and types of arguments to the function. The VARARGS and ARGSUSED comments can be used to specify features of the library functions.

The *lint* library files are processed almost exactly like ordinary source files. The only difference is that functions which are defined on a library file but are not used on a source file do not result in messages. The *lint* program does not simulate a full library search algorithm and will print messages if the source files contain a redefinition of a library routine.

By default, *lint* checks the programs it is given against a standard library file which contains descriptions of the programs which are normally loaded when a C language program is run. When the **−p** option is used, another file is checked containing descriptions of the standard library routines which are expected to be portable across various machines. The **−n** option can be used to suppress all library checking.

Directives

The alternative to using options to suppress *lint*'s comments about problem areas is to use directives. Directives appear in the source code in the form of code comments. *Lint* recognizes five directives.

/*NOTREACHED*/	stops an unreachable code comment about the next line of code.
/*NOSTRICT*/	stops *lint* from strictly type checking the next expression.
/*ARGSUSED*/	stops a comment about any unused parameters for the following function.
/*VARRARGS*n*/	stops *lint* from reporting variable numbers of parameters in calls to a function. The function's declaraction follows this comment. The first *n* parameters must be present in each call to the function; *lint* comments if they aren't. if "/*VARARGS*/" appears without the *n*, none of the parameters need be present.
/*LINTLIBRARY*/	must be placed at the beginning of a file. This directive tells *lint* that the file is a library file and to suppress comments about the unused functions. *Lint* objects if other files redefine routines that are found there.

Index

a

ARGSUSED 14

c

C function libraries 13
.c suffix required on files 12
code unreachable 5
command syntax, lint 12
compiler used to detect errors 1
constructions, strange 10

d

detecting errors with compiler 1
directives, lint 14

e

error detection 1
error limit 1
error messages, suppressing 3
external symbols 9
external variables and functions 4

f

filename suffix .c required 12
function libraries, C 13
function return value 5
functions, unused 3

i

infinite loop 5

l

libraries, C function 13
library file processing by lint 13
limit, error 1
lint directives 14
lint, purpose of 2
LINTLIBRARY 14
loop not entered from top 5

m

member type consistency 7

n

NOSTRICT 14
NOTREACHED 14

o

operator precedence confusion 11

p

pointer alignment 9
portability 8
precedence confusion 11
problem detection 2
purpose of lint 2

r

redefining variables 11

s

strange constructions 10
structure type references 8
subtle errors 2
suppressing error messages 3
symbols, external 9
syntax, lint command 12

t

type matching, neglected by compiler 6
types 7

u

undefined return value 6
uninitialized external variables 9
union type references 8
unreachable code 5
unsigned treated as signed 10
unused variables and functions 3
using compiler to detect errors 1
using values before they are set 4

v

value set but not used 4
value used but not yet set 4
variable type consistency 7
variables redefinition 11
variables, unused 3
VARRARGSn 14

Table of Contents

Ratfor: A Preprocessor for a National FORTRAN

Introduction 2
Language Description 3
Design 3
Statement Grouping 4
The "else" Clause 5
Nested if's 6
If-else Ambiguity 8
The "switch" Statement 9
The "do" Statement 9
"Break" and "next" 11
The "while" Statement 11
The "for" Statement 13
The "repeat-until" statement 15
More on break and next 16
The "return" Statement 16
Cosmetics 17
Free-form Input 17
Translation Services 18
The "define" Statement 20
The "include" Statement 21
Pitfalls, Botches, Blemishes and other Failings 21
Experience 22
Good Things 22
Bad Things 23
Conclusions 24
Usage on HP-UX 24

Ratfor: A Preprocessor for a Rational FORTRAN

Although FORTRAN is not a pleasant language to use, its universality and relative efficiency maintain its position in the computer market. The Ratfor language, by providing control flow statements, attempts to conceal the main deficiencies of FORTRAN while retaining its desirable qualities. The Ratfor preprocessor converts input code into FORTRAN output code. The facilities provided include:

- Statement grouping
- `If-else` and `switch` for decision-making
- `While`, `for`, `do`, and `repeat-until` for looping
- `Break` and `next` for controlling loop exits
- Free-form input such as multiple statements/lines, and automatic continuation
- Simple comment convention
- Translation of >, >=, etc., into .gt., .ge., etc.
- `Return` function for functions
- `Define` statement for symbolic parameters
- `Include` statement for including source files.

Introduction

Most programmers agree that FORTRAN is an unpleasant language to program in, yet there are many occasions when they are forced to use it, especially when FORTRAN is the only language thoroughly supported on the local computer, or the application requires intensive computation.

FORTRAN's worst deficiency is probably in control flow statements, conditional branches and loops, that express the logic of program flow. For example, FORTRAN's primitive conditional statements force the user into at least two statement numbers and two implied `GOTO`s to handle a single arithmetic `IF`. This leads to unintelligible code that is eschewed by good programmers.

The Logical `IF` is better, in that the test part can be stated clearly, but hopelessly restrictive because the statement that follows the *IF* can only be one FORTRAN statement (with some **further** restrictions!).

The result of these failings is that FORTRAN programs must be written with numerous labels and branches. The resulting code is particularly difficult to read and understand, and thus hard to debug and modify.

When one is faced with an unpleasant language, a useful technique is to define a new language that overcomes the deficiencies, and to translate it into the unpleasant one with a preprocessor. This is the approach taken with Ratfor (The preprocessor idea is not new, and FORTRAN preprocessors are widely used).

Language Description

Design

Ratfor attempts to retain the merits of FORTRAN (universality, portability, efficiency) while hiding the worst FORTRAN inadequacies. The language **is** FORTRAN except for two aspects. First, since control flow is central to any program, regardless of the specific application, the primary task of Ratfor is to conceal this part of FORTRAN from the user, by providing decent control flow structures. These structures are sufficient and comfortable for structured programming in the narrow sense of programming without *GOTO*'s. Second, since the preprocessor must examine an entire program to translate the control structure, it is possible at the same time to clean up many of the "cosmetic" deficiencies of FORTRAN, and thus provide a language which is easier and more pleasant to read and write.

Beyond these two aspects — control flow and cosmetics — Ratfor does nothing about the host of other weaknesses of FORTRAN. Although it would be straightforward to extend it to provide character strings, for example, they are not needed by everyone, and of course the preprocessor would be harder to implement. Throughout, the design principle which has determined what should be in Ratfor and what should not has been *Ratfor* **doesn't** know any FORTRAN. Any language feature which would require that Ratfor really understand FORTRAN has been omitted. We will return to this point in the section on implementation.

Even within the confines of control flow and cosmetics, we have attempted to be selective in what features to provide. The intent has been to provide a small set of the most useful constructs, rather than to throw in everything that has ever been thought useful by someone.

The rest of this section contains an informal description of the Ratfor language. The control flow aspects will be quite familiar to readers used to languages like Algol, PL/I, Pascal, etc., and the cosmetic changes are equally straightforward. We shall concentrate on showing what the language looks like.

Statement Grouping

FORTRAN provides the Block If statement to group statements together. However, the keyword approach can be cumbersome. The standard construction "if a condition is true, do this group of things," for example,

```
if (x > 100)
   { call error("x>100"); err = 1; return }
```

would be written in FORTRAN as

```
if (x .gt. 100) then
      call error("x > 100")
      err = 1
      return
endif
```

A group of statements can be treated as a unit by enclosing them in the braces { and }. This is true throughout the language: wherever a single Ratfor statement can be used, there can be several enclosed in braces. (Braces seem clearer and less obtrusive than *begin* and *end* or *do* and *end*, and of course *do* and *end* already have FORTRAN meanings.)

Cosmetics contribute to the readability of code, and thus to its understandability. The character ">" is clearer than *".GT."*, so Ratfor translates it appropriately, along with several other similar shorthands.

Ratfor is a free-form language: statements may appear anywhere on a line, and several may appear on one line if they are separated by semicolons. The example above could also be written as

```
if (x > 100) {
   call error("x>100")
   err = 1
   return
}
```

In this case, no semicolon is needed at the end of each line because Ratfor assumes there is one statement per line unless told otherwise.

Of course, if the statement that follows the `if` is a single statement (Ratfor or otherwise), no braces are needed:

```
if (y <= 0.0 & z <= 0.0)
   write(6, 20) y, z
```

No continuation need be indicated because the statement is clearly not finished on the first line. In general Ratfor continues lines when it seems obvious that they are not yet done. (The continuation convention is discussed in detail later.)

Although a free-form language permits wide latitude in formatting styles, it is wise to pick one that is readable, then stick to it. In particular, proper indentation is vital, to make the logical structure of the program obvious to the reader.

The "else" Clause

Ratfor provides an "*else*" statement to handle the construction "if a condition is true, do this thing, otherwise do that thing."

```
if (a <= b)
     { sw = 0; write(6, 1) a, b }
else
     { sw = 1; write(6, 1) b, a }
```

This writes out the smaller of *a* and *b*, then the larger, and sets *sw* appropriately.

As before, if the statement following an *if* or an *else* is a single statement, no braces are needed:

```
if (a <= b)
   sw = 0
else
   sw = 1
```

The syntax of the *if* statement is

```
if (<legal FORTRAN condition>)
   Ratfor statement
else
   Ratfor statement
```

where the *else* part is optional. The <legal FORTRAN condition> is anything that can legally go into a FORTRAN Logical `IF`. Ratfor does not check this clause, since it does not know enough FORTRAN to know what is permitted. The *Ratfor statement* is any Ratfor or FORTRAN statement, or any collection of them in braces.

Nested if's

Since the statement that follows an *if* or an *else* can be any Ratfor statement, this leads immediately to the possibility of another *if* or *else*. As a useful example, consider this problem:

The variable f is to be set to -1 if x is less than zero, to $+1$ if x is greater than 100, and to 0 otherwise. In Ratfor, we write

```
if (x < 0)
    f = -1
else if (x > 100)
    f = +1
else
    f = 0
```

Here the statement after the first `else` is another `if-else`. Logically it is just a single statement, although it is rather complicated.

This code says what it means. Any version written in straight Fortran will necessarily be indirect because Fortran does not let you say what you mean. And as always, clever shortcuts may turn out to be too clever to understand a year from now.

Following an *else* with an *if* is one way to write a multi-way branch in Ratfor. In general the structure

```
if (...)
    - - -
else if (...)
    - - -
else if (...)
    - - -
...
else
  ---
```

provides a way to specify the choice of exactly one of several alternatives. (Ratfor also provides a *switch* statement which does the same job in certain special cases; in more general situations, we have to make do with spare parts.) The tests are laid out in sequence, and each one is followed by the code associated with it. Read down the list of decisions until one is found that is satisfied. The code associated with this condition is executed, and then the entire structure is finished. The trailing *else* part handles the "default" case, where none of the other conditions apply. If there is no default action, this final *else* part is omitted:

```
if (x < 0)
    x = 0
else if (x > 100)
    x = 100
```

If-else Ambiguity

There is one thing to notice about complicated structures involving nested *if*s and *else*s. Consider

```
if (x > 0)
        if (y > 0)
            write(6, 1) x, y
        else
            write(6, 2) y
```

There are two *if*s and only one *else*. Which *if* does the *else* go with?

This is a genuine ambiguity in Ratfor, as it is in many other programming languages. The ambiguity is resolved in Ratfor (as elsewhere) by saying that in such cases the *else* goes with the closest previous un-*else*ed *if*. Thus in this case, the *else* goes with the inner *if*, as we have indicated by the indentation.

It is a wise practice to resolve such cases by explicit braces, just to make your intent clear. In the case above, we would write

```
if (x > 0) {
    if (y > 0)
        write(6, 1) x, y
    else
        write(6, 2) y
}
```

which does not change the meaning, but leaves no doubt in the reader's mind. If we want the other association, we **must** write

```
if (x > 0) {
    if (y > 0)
        write(6, 1) x, y
}
else
    write(6, 2) y
```

The "switch" Statement

The *switch* statement provides a clean way to express multi-way branches which branch on the value of some integer-valued expression. The syntax is

```
switch (<expression>) {
    case <expr1>:
        statements
    case <expr2>, <expr> :
        statements
    ...
 default:
    statements
}
```

Each *case* is followed by a list of comma-separated integer expressions. The <expression> inside *switch* is compared against the case expressions <expr1>, <expr2>, and so on in turn until one matches, at which time the statements following that *case* are executed. If no cases match <expression>, and there is a *default* section, the statements with it are done; if there is no *default*, nothing is done. In all situations, as soon as some block of statements is executed, the entire *switch* is exited immediately. (Readers familiar with C should beware that this behavior is not the same as the C *switch*.)

The "do" Statement

The *do* statement in Ratfor is quite similar to the DO statement in FORTRAN, except that it uses no statement number. The statement number, after all, serves only to mark the end of the DO, and this can be done just as easily with braces. Thus

```
do i = 1, n {
    x(i) = 0.0
    y(i) = 0.0
    z(i) = 0.0
}
```

is the same as

```
    do 10 i = 1, n
        x(i) = 0.0
        y(i) = 0.0
        z(i) = 0.0
10  continue
```

The syntax is:

```
do <legal FORTRAN text>
        Ratfor statement
```

The part that follows the keyword *do* has to be something that can legally go into a FORTRAN `DO` statement. Thus if a local version of FORTRAN allows `DO` limits to be expressions (which is not currently permitted in ANSI FORTRAN), they can be used in a Ratfor *do*.

The *Ratfor statement* part will often be enclosed in braces, but as with the *if*, a single statement need not have braces around it. This code sets an array to zero:

```
do i = 1, n
    x(i) = 0.0
```

Slightly more complicated,

```
do i = 1, n
    do j = 1, n
        m(i, j) = 0
```

sets the entire array *m* to zero, and

```
do i = 1, n
    do j = 1, n
        if (i < j)
            m(i, j) = -1
        else if (i == j)
            m(i, j) = 0
        else
            m(i, j) = +1
```

sets the upper triangle of *m* to -1, the diagonal to zero, and the lower triangle to $+1$. (The operator == is "equals"; that is, ".EQ.".) In each case, the statement that follows the *do* is logically a **single** statement, even though complicated, and thus needs no braces.

"Break" and "next"

Ratfor provides a statement for leaving a loop early, and one for beginning the next iteration. `Break` causes an immediate exit from the `do`; in effect it is a branch to the statement **after** the `do`. `Next` is a branch to the bottom of the loop, so it causes the next iteration to be done. For example, this code skips over negative values in an array:

```
do i = 1, n {
    if (x(i) < 0.0)
        next
    <process positive element>
}
```

`Break` and `next` also work in the other Ratfor looping constructions discussed in the next few sections.

`Break` and *next* can be followed by an integer to indicate breaking or iterating that level of enclosing loop; thus

```
break 2
```

exits from two levels of enclosing loops, and `Break 1` is equivalent to `break`. `next 2` iterates the second enclosing loop. (Realistically, multi-level `breaks` and `nexts` are not likely to be much used because they lead to code that is hard to understand and somewhat risky to change.)

The "while" Statement

A serious problem with the `DO` statement is that it encourages that a program be written in terms of an arithmetic progression with small positive steps, even though that may not be the best way to write it. If code has to be contorted to fit the requirements imposed by the FORTRAN `DO`, it is that much harder to write and understand.

To overcome these difficulties, Ratfor provides a `while` statement, which is simply a loop: "while some condition is true, repeat this group of statements". It has no preconceptions about why one is looping. For example, this routine to compute sin(x) by the Maclaurin series combines two termination criteria.

```
real function sin(x, e)
    # returns sin(x) to accuracy e, by
    # sin(x) = x - x**3/3! + x**5/5! - ...
    sin = x
    term = x
    i = 3
    while (abs(term)>e & i<100) {
        term = -term * x**2 / float(i*(i-1))
        sin = sin + term
        i = i + 2
    }
    return
    end
```

Notice that if the routine is entered with *term* already smaller than *e*, the loop will be done **zero times**, that is, no attempt will be made to compute `x**3` and thus a potential underflow is avoided. Since the test is made at the top of a `while` loop instead of the bottom, a special case disappears: the code works at one of its boundaries. (The test *i<100* is the other boundary, making sure the routine stops after some maximum number of iterations.)

As an aside, a sharp character "*#*" in a line marks the beginning of a comment; the rest of the line is comment. Comments and code can co-exist on the same line — one can make marginal remarks, which is not possible with FORTRAN's "C in column 1" convention. Blank lines are also permitted anywhere; they should be used to emphasize the natural divisions of a program.

The syntax of the `while` statement is

`while` (*legal* FORTRAN condition)
Ratfor statement

As with the *if*, (legal FORTRAN condition) is something that can go into a FORTRAN Logical `IF`, and *Ratfor* statement is a single statement, which may be multiple statements in braces.

The `while` encourages a style of coding not normally practiced by FORTRAN programmers. For example, suppose *nextch* is a function which returns the next input character both as a function value and in its argument. Then a loop to find the first non-blank character is just

```
while (nextch(ich) == iblank)
    ;
```

A semicolon by itself is a null statement, which is necessary here to mark the end of the `while`; if it were not present, the `while` would control the next statement. When the loop is broken, `ich` contains the first non-blank. Of course the same code can be written in FORTRAN as

```
100 if (nextch(ich) .eq. iblank) goto 100
```

but many FORTRAN programmers (and a few compilers) believe this line is illegal. The language at one's disposal strongly influences how one thinks about a problem.

The "for" Statement

The `for` statement is another Ratfor loop, which attempts to carry the separation of loop-body from reason-for-looping a step further than the `while`. A `for` statement allows explicit initialization and increment steps as part of the statement. For example, a `DO` loop is just

```
for (i = 1; i <= n; i = i + 1) ...
```

This is equivalent to

```
i = 1
while (i <= n) {
    ...
    i = i + 1
}
```

The initialization and increment of *i* have been moved into the *for* statement, making it easier to see at a glance what controls the loop.

The *for* and *while* versions have the advantage that they will be done zero times if *n* is less than 1.

The loop of the sine routine in the previous section can be rewritten with a *for* as

```
for (i=3; abs(term) > e & i < 100; i=i+2) {
    term = -term * x**2 / float(i*(i-1))
    sin = sin + term
}
```

The syntax of the *for* statement is

```
for (<init>; <condition>; <increment>)
    Ratfor statement
```

<init> is any single FORTRAN statement that is executed once before the loop begins.

<increment> is any single FORTRAN statement, that gets done at the end of each pass through the loop, before the test.

<condition> is, again, anything that is legal in a logical `IF`.

Any of <init>, <condition>, and <increment> can be omitted, although the semicolons **must** always be present. A non-existent <condition> is treated as always true, so *for(;;)* is an indefinite repeat (but see the *repeat-until* in the next section).

The *for* statement is particularly useful for backward loops, chaining along lists, loops that might be done zero times, and similar things which are hard to express with a `DO` statement, and obscure to write out with `IF`s and `GOTO`s. For example, here is a backwards `DO` loop to find the last non-blank character on a card:

```
for (i = 80; i > 0; i = i - 1)
    if (card(i) != blank)
        break
```

(`!=` is the same as `.NE.`). The code scans the columns from 80 through to 1. If a non-blank is found, the loop is immediately broken *break and next* work in *for*s and *while*s just as in *do*s). If *i* reaches zero, the card is all blank.

This code is rather nasty to write with ANSI FORTRAN `DO`, since proper conditions must explicitly set up when we fall out of the loop (forgetting this is a common error). Thus:

```
      DO 10 I = 80, 1, -1
            IF (CARD(I) .NE. BLANK) GO TO 11
10    CONTINUE
          I = 0
11    ...
```

The version that uses the *for* handles the termination condition properly for free; *i* is zero when we fall out of the *for* loop.

The increment in a *for* need not be an arithmetic progression; the following program walks along a list (stored in an integer array *ptr*) until a zero pointer is found, adding up elements from a parallel array of values:

```
sum = 0.0
for (i = first; i > 0; i = ptr(i))
    sum = sum + value(i)
```

Notice that the code works correctly if the list is empty. Again, placing the test at the top of a loop instead of the bottom eliminates a potential boundary error.

The "repeat-until" statement

In spite of the dire warnings, there are times when one really needs a loop that tests at the bottom after one pass through. This service is provided by the *repeat-until*:

```
repeat
    Ratfor statement
until (legal FORTRAN condition)
```

The *Ratfor* statement part is done once, then the condition is evaluated.

- If it is true, the loop is exited.
- If it is false, another pass is made.

The *until* part is optional, so a bare *repeat* is the cleanest way to specify an infinite loop.

Of course such a loop must ultimately be broken by some transfer of control such as *stop*, *return*, or *break*, or an implicit stop such as running out of input with a `READ` statement.

It is a matter of observed fact that the *repeat-until* statement is **much** less used than the other looping constructions; in particular, it is typically outnumbered ten to one by *for* and *while*. Be cautious about using it, for loops that test only at the bottom often don't handle null cases well.

More on *break* and *next*

Break exits immediately from *do*, *while*, *for*, and *repeat-until*. *Next* goes to the test part of *do*, *while* and *repeat-until*, and to the increment step of a *for*.

The "return" Statement

The standard FORTRAN mechanism for returning a value from a function uses the name of the function as a variable. The variable is assigned by the program, and the last value stored in it is the function value upon return. For example, here is a routine *equal* which returns 1 if two arrays are identical, and zero if they differ. The array ends are marked by the special value -1.

```
# equal - compare str1 to str2;
#
return 1 if equal, 0 if not
    integer function equal(str1, str2)
    integer str1(100), str2(100)
    integer i
    for (i = 1; str1(i) == str2(i); i = i + 1)
        if (str1(i) == -1) {
           equal = 1
           return
        }
    equal = 0
    return
    end
```

In many languages (e.g., PL/I) one instead says

```
return ( <expression> )
```

to return a value from a function. Since this is often clearer, Ratfor provides such a *return* statement. In a function *f*, *return (*expression*)* is equivalent to

```
{ F = <expression>; <return> }
```

For example, here is *equal* again:

```
# equal _ compare str1 to str2;
#
return 1 if equal, 0 if not
    integer function equal(str1, str2)
    integer str1(100), str2(100)
    integer i
    for (i = 1; str1(i) == str2(i); i = i + 1)
        if (str1(i) == -1)
        return(1)
    return(0)
    end
```

If there is no parenthesized expression after *return*, a normal `RETURN` is made. (Another version of *equal* is presented shortly.)

Cosmetics

As previously stated, the visual appearance of a language has a substantial effect on how easy it is to read and understand programs. Accordingly, Ratfor provides a number of cosmetic facilities which may be used to make programs more readable.

Free-form Input

Statements can be placed anywhere on a line. Long statements are continued automatically, as are long conditions in *if*, *while*, *for*, and *until*. Blank lines are ignored. Multiple statements may appear on one line if they are separated by semicolons. No semicolon is needed at the end of a line if Ratfor can make some reasonable guess about whether the statement ends there. Lines ending with any of the characters

```
=    +    -    *    ,    |    &    (    _
```

are assumed to be continued on the next line. Underscores are discarded wherever they occur; all others remain as part of the statement.

Any statement that begins with an all-numeric field is assumed to be a FORTRAN label, and placed in columns 1-5 upon output. Thus

```
write(6, 100); 100 format("hello")
```

is converted into

```
    write(6, 100)
100 format(5hhello)
```

Translation Services

Text enclosed in matching single or double quotes is converted to `nH...` but is otherwise unaltered (except for formatting — it may get split across card boundaries during the reformatting process). Within quoted strings, the backslash (\) serves as an escape character: the next character is taken literally. This provides a way to get quotes (and of course the backslash itself) into quoted strings:

```
"\\\'"
```

is a string containing a backslash and an apostrophe. (This is **not** the standard convention of doubled quotes, but it is easier to use and more general.)

Any line that begins with the character (%) is left absolutely unaltered except for stripping off the (%) and moving the line one position to the left. This is useful for inserting control cards, and other things that should not be transmogrified (like an existing FORTRAN program). Use (%) only for ordinary statements; not for the condition parts of *if*, *while*, etc.; or the output may be positioned incorrectly.

The following character translations are made, except within single or double quotes or on a line beginning with a percent sign (%).

Input	Translated output
==	.eq.
!=	.ne.
>	.gt.
>=	.ge.
<	.lt.
<=	.le.
&	.and.
\|	.or.
!	.not.
^	.not.

In addition, the following translations are provided for input devices with restricted character sets.

[	{
]	}
$(	{
$)	}

The "define" Statement

Any string of alphanumeric characters can be defined as a name; thereafter, whenever that name occurs in the input (delimited by non-alphanumerics) it is replaced by the rest of the definition line. (Comments and trailing white spaces are stripped off). A defined name can be arbitrarily long, and must begin with a letter.

`Define` is typically used to create symbolic parameters:

```
define ROWS 100
define COLS 50
dimension a(ROWS), b(ROWS, COLS)
    if (i > ROWS | j > COLS) ...
```

Alternately, definitions can be written as

```
define(ROWS, 100)
```

In this case, the defining text is everything after the comma up to the balancing right parenthesis; this allows multi-line definitions.

It is generally a wise practice to use symbolic parameters for most constants, to help make clear the function of what would otherwise be mysterious numbers. As an example, here is the routine *equal* again, this time with symbolic constants.

```
define YES 1
define NO 0
define EOS -1
define ARB 100
# equal - compare str1 to str2;
#
return YES if equal, NO if not
    integer function equal(str1, str2)
    integer str1(ARB), str2(ARB)
    integer i
    for (i = 1; str1(i) == str2(i); i = i + 1)
        if (str1(i) == EOS)
        return(YES)
    return(NO)
    end
```

The "include" Statement

The statement

```
include file
```

inserts the file found on input stream `file` into the Ratfor input in place of the `include` statement. The standard usage is to place `COMMON` blocks on a file, and *include* that file whenever a copy is needed:

```
subroutine x
    include commonblocks
    ...
    end
subroutine y
    include commonblocks
    ...
    end
```

This ensures that all copies of the `COMMON` blocks are identical

Pitfalls, Botches, Blemishes and other Failings

Ratfor catches certain syntax errors, such as missing braces, `else` clauses without an `if`, and most errors involving missing parentheses in statements. Beyond that, since Ratfor knows no FORTRAN, any errors you make will be reported by the FORTRAN compiler, so you will from time to time have to relate a FORTRAN diagnostic back to the Ratfor source.

Keywords are reserved. Using *if*, *else*, etc., as variable names will typically wreak havoc.

Don't leave spaces in keywords. Don't use the Arithmetic `IF`.

The FORTRAN `nH` convention is not recognized anywhere by Ratfor; use quotes instead.

Experience

Good Things

"It's so much better than FORTRAN" is the most common response of users when asked how well Ratfor meets their needs. Although cynics might consider this to be vacuous, it does seem to be true that decent control flow and cosmetics converts FORTRAN from a bad language into quite a reasonable one, assuming that FORTRAN data structures are adequate for the task at hand.

Although there are no quantitative results, users feel that coding in Ratfor is at least twice as fast as in FORTRAN. More important, debugging and subsequent revision are much faster than in FORTRAN. Partly this is simply because the code can be *read*. The looping statements which test at the top instead of the bottom seem to eliminate or at least reduce the occurrence of a wide class of boundary errors. And of course it is easy to do structured programming in Ratfor; this self-discipline also contributes markedly to reliability.

One interesting and encouraging fact is that programs written in Ratfor tend to be as readable as programs written in more modern languages like Pascal. Once one is freed from the shackles of FORTRAN's clerical detail and rigid input format, it is easy to write code that is readable, even esthetically pleasing. For example, here is a Ratfor implementation of a linear table search:

```
A(m+1) = x
for (i = 1; A(i) != x; i = i + 1)
    ;
if (i > m) {
    m = i
    B(i) = 1
}
else
    B(i) = B(i) + 1
```

Bad Things

The biggest single problem is that many FORTRAN syntax errors are not detected by Ratfor but by the local FORTRAN compiler. The compiler then prints a message in terms of the generated FORTRAN, and in a few cases this may be difficult to relate back to the offending Ratfor line, especially if the implementation conceals the generated FORTRAN. This problem could be dealt with by tagging each generated line with some indication of the source line that created it, but this is inherently implementation-dependent, so no action has yet been taken. Error message interpretation is actually not so arduous as might be thought. Since Ratfor generates no variables, only a simple pattern of `IF`s and `GOTO`s, data-related errors like missing `DIMENSION` statements are easy to find in the FORTRAN. Furthermore, there has been a steady improvement in Ratfor's ability to catch trivial syntactic errors like unbalanced parentheses and quotes.

There are a number of implementation weaknesses that are a nuisance, especially to new users. For example, keywords are reserved. This rarely makes any difference, except for those hardy souls who want to use an Arithmetic `IF`. A few standard FORTRAN constructions are not accepted by Ratfor, and this is perceived as a problem by users with a large corpus of existing FORTRAN programs. Protecting every line with a (%) is not really a complete solution, although it serves as a stop-gap. The best long-term solution is provided by the program *struct*, which converts arbitrary FORTRAN programs into Ratfor.

Users who export programs often complain that the generated FORTRAN is "unreadable" because it is not tastefully formatted and contains extraneous `CONTINUE` statements. To some extent this can be ameliorated (Ratfor now has an option to copy Ratfor comments into the generated FORTRAN), but it has always seemed that effort is better spent on the input language than on the output esthetics.

One final problem is partly attributable to success; since Ratfor is relatively easy to modify, there are now several dialects of Ratfor. Fortunately, so far most of the differences are in character set, or in invisible aspects like code generation.

Conclusions

Ratfor demonstrates that with modest effort it is possible to convert FORTRAN from a bad language into quite a good one. A preprocessor is clearly a useful way to extend or ameliorate the facilities of a base language.

When designing a language, it is important to concentrate on the essential requirement of providing the user with the best language possible for a given effort. One must avoid throwing in "features"; things which the user may trivially construct within the existing framework.

One must also avoid getting sidetracked on irrelevancies. For instance it seems pointless for Ratfor to prepare a neatly formatted listing of either its input or its output. The user is presumably capable of the self-discipline required to prepare neat input that reflects his thoughts. It is much more important that the language provide free-form input so he **can** format it neatly. No one should read the output anyway except in the most dire circumstances.

Usage on HP-UX

The program *ratfor* is the basic translator; it takes either a list of file names or the standard input and writes FORTRAN on the standard output. Options include `-6x`, which uses `x` as a continuation character in column 6 (HP-UX uses & in column 1), and $-C$, which causes Ratfor comments to be copied into the generated FORTRAN.

Index

b

backward loops 14
braces 4, 8
branching, multi-way 7, 9
break (exit from do) 11
break (exit from do, while, for, and repeat-until) 16

c

case statement used with switch statement, example 9
chaining-along lists 14
character translations 19
comment lines 12
common blocks on a file 21
continue line on next line 17
control-flow 3
copies of common blocks 21
cosmetics 17

d

define statement 20
do, exit from 11
do statement 9, 10

e

else clause 5
exit from do 11

f

for statement 13–15
free-form input 17

g

grouping statements 4

h

HP-UX usage 24

i

if 2, 5
if-else ambiguity 8
if-else structure 5
ifs, nested 6
include files 21
infinite loop 15

l

limitations 21
line continued on next line 17
logical IF 2
loop, infinite 15
loops executed zero times 14

m

multi-level break or next 11
multi-way branching 7, 9
multiple statements on a line 4

n

nested ifs 6
next (branch to bottom of do or other loop) 11
numeric field at beginning of statement 18

r

ratfor on HP-UX 24
repeat-until statement 15
return statement 16
return value from a function 16

s

statement grouping 4
string comparer example program 16, 17, 20
switch statement 7, 9
symbolic constants, used to clarify functions 20

t

text translation 18
translation of text characters 18

u

underflow prevention 12, 13
underscores discarded 17
until statement 15

v

visual appearance 17

w

weaknesses 3
while statement 11–13

Table of Contents

The M4 Macro Processor

General 1
Usage 4
Defining Macros 4
Arguments 8
Arithmetic Functions 9
File Manipulation 10
System Command 11
Conditionals 11
String Manipulation 12
Printing 13

The M4 Macro Processor

General

The *m4* macro processor is a front end for rational FORTRAN (*Ratfor*) and the C programming languages. The "#define" statement in C language and the analogous "define" in *Ratfor* are examples of the basic facility provided by any macro processor.

At the beginning of a program, a symbolic name or symbolic constant can be defined as a particular string of characters. The compiler will then replace later unquoted occurrences of the symbolic name with the corresponding string. Besides the straightforward replacement of one string of text by another, the *m4* macro processor provides the following features:

- arguments
- arithmetic capabilities
- file manipulation
- conditional macro expansion
- string and substring functions.

The basic operation of *m4* is to read every alphanumeric token (string of letters and digits) input and determine if the token is the name of a macro. The name of the macro is replaced by its defining text, and the resulting string is pushed back onto the input to be re-scanned. Macros may be called with arguments. The arguments are collected and substituted into the right places in the defining text before the defining text is re-scanned.

The user also has the capability to define new macros. Built-ins and user-defined macros work exactly the same way except that some of the built-in macros have side effects on the state of the process. A list of 32 built-in macros provided by the *m4* macro processor can be found in Table 1.

Table 1. Built-in Macros

Macro Name	Function
changequote	Restores original characters or sets new quote characters.
changecom	Changes left and right comment markers from the default # and new line.
decr	Returns the value of its argument decremented by 1.
define	Defines new macros.
defn	Returns the quoted definition of its argument(s).
divert	Diverts output to 1-out-of-10 diversions.
divnum	Returns the number of the currently active diversion.
dnl	Reads and discards characters up to and including the next new line.
dumpdef	Dumps the current names and definitions of items named as arguments.
errprint	Prints its arguments on the standard error file.
eval	Prints arbitrary arithmetic on integers.
ifdef	Determines if a macro is currently defined.
ifelse	Performs arbitrary conditional testing.
include	Returns the contents of the file named in the argument. A fatal error occurs if the file name cannot be accessed.
incr	Returns the value of its argument incremented by 1.
index	Returns the position where the second argument begins in the first argument.

Table 1. (Continued)

Macro Name	Function
len	Returns the number of characters that makes its argument.
m4exit	Causes immediate exit from *m4*.
m4wrap	Pushes the exit code back at final EOF.
maketemp	Facilitates making unique file names.
popdef	Removes current definition of its argument(s) exposing any previous definitions.
pushdef	Defines new macros but saves any previous definition.
shift	Returns all arguments of shift except the first argument.
sinclude	Returns the contents of the file named in the arguments. The macro remains silent and continues if the file is inaccessible.
substr	Produces substrings of strings.
syscmd	Executes the HP-UX System command given in the first argument.
sysval	Is the return code from the last call to **syscmd**.
traceoff	Turns macro trace off.
traceon	Turns the macro trace on.
translit	Performs character transliteration.
undefine	Removes user-defined or built-in macro definitions.
undivert	Discards the diverted text.

Usage

To use the *m4* macro processor, input the following command:

m4 [*optional files*]

Each argument file is processed in order. If there are no arguments or if an argument is "-", the standard input is read at that point. The processed text is written on the standard output which may be captured for subsequent processing with the following command:

m4 [*files*] > *outputfile*

Defining Macros

The primary built-in function of *m4* is **define.** **Define** is used to define new macros. The following input:

define(*name,stuff*)

causes the string *name* to be defined as *stuff*. All subsequent occurrences of *name* will be replaced by *stuff*. *Name* must be alphanumeric and must begin with a letter (the underscore counts as a letter). *Stuff* is any text that contains balanced parentheses. *Stuff* may stretch over multiple lines. Thus, as a typical example:

```
define(N, 100)
. . .
if (i > N)
```

defines N to be 100 and uses the symbolic constant N in a later **if** statement.

The left parenthesis must immediately follow the word **define** to signal that **define** has arguments. If a user-defined macro or function name is not followed immediately by "(", it is assumed to have no arguments. Macro calls have the following general form:

name(*arg1,arg2,...argn*)

A macro name is only recognized as such if it appears surrounded by non-alphanumerics. Using the following example:

```
define(N, 100)
...
if (NNN > 100)
```

the variable NNN is absolutely unrelated to the defined macro N, even though the variable contains a lot of Ns.

Macros may be defined in terms of other names. For example:

```
define(N, 100)
define(M, N)
```

defines both M and N to be 100. If N is redefined and subsequently changes, M retains the value of 100, not N.

The *m4* macro processor expands macro names into their defining text as soon as possible. The string N is immediately replaced by 100. Then the string M is also immediately replaced by 100. The overall result is the same as using the following input in the first place:

```
define(M, 100)
```

The order of the definitions can be interchanged as follows:

```
define(M, N)
define(N, 100)
```

Now M is defined to be the string N, so when the value of M is requested later, the result is the value of N at that time (because the M will be replaced by N, which will be replaced by 100).

The more general solution is to delay the expansion of the arguments of **define** by quoting them. Any text surrounded by left and right single quotes is not expanded immediately but has the quotes stripped off.

NOTE

The direction of the single quote marks is important. The left (opening) quote is different from the right (closing) quote. See **changequote** if this is a problem on your terminal.

The value of a quoted string is the string stripped of the quotes. If the input is:

```
define(N, 100)
define(M, 'N')
```

the quotes around the N are stripped off as the argument is being collected. The result of using quotes is to define M as the string N, not 100. The general rule is that *m4* always strips off one level of single quotes whenever it evaluates something. This is true even outside of macros. If the word **define** is to appear in the output, the word must be quoted in the input as follows:

```
'define' = 1
```

Another example of using quotes is redefining N. To redefine N, the evaluation must be delayed by quoting:

```
define(N, 100)
...
define('N', 200)
```

In *m4*, it is often wise to quote the first argument of a macro. The following example will not redefine N:

```
define(N, 100)
...
define(N, 200)
```

The N in the second definition is replaced by 100. The result is equivalent to the following statement:

```
define(100,200)
```

This statement causes an error since only things that look like names can be defined.

If left and right single quotes are not convenient for some reason, the quote characters can be changed using the **changequote** function. The following example substitutes the opening and closing square brackets for the opening and closing single quote characters:

```
changequote([, ])
```

The original characters can be restored by using **changequote** without arguments as follows:

```
changequote
```

There are two additional built-ins related to **define**. The **undefine** macro removes the definition of some macro or function as follows:

```
undefine('N')
```

The macro removes the definition of N. Built-ins can be removed with **undefine**, as follows:

```
undefine('define')
```

But once removed, the definition cannot be re-used.

The function **ifdef** provides a way to determine if a macro is currently defined. Depending on the system, a definition appropriate for the particular machine can be made as follows:

```
ifdef('S310', 'define(wordsize,16)')
ifdef('S550', 'define(wordsize,32)')
```

Remember to use the quotes.

The **ifdef** macro actually permits three arguments. If the first argument is defined, the value of **ifdef** is the second argument. If the first argument is not defined, the value of **ifdef** is the third argument. If there is no third argument, the value of **ifdef** is null. Example:

```
ifdef('hpux', on HPUX, not on HPUX)
```

Arguments

So far, the simplest form of macro processing has been discussed, which is replacing one string by another (fixed) string. User-defined macros may also have arguments, so different invocations can have different results. Within the replacement text for a macro (the second argument of its **define**), any occurrence of **$n** is replaced by the **n**th argument when the macro is actually used. Thus, the macro **bump** defined as:

```
define(bump, $1 = $1 + 1)
```

generates code to increment its argument by 1. The 'bump(x)' statement is equivalent to 'x = x + 1'.

A macro can have as many arguments as needed, but only the first nine are accessible (**$1** through **$9**). The macro name is **$0**, although that is less commonly used. Arguments that are not supplied are replaced by null strings, so a macro can be defined which simply concatenates its arguments like this:

```
define(cat, $1$2$3$4$5$6$7$8$9)
```

Thus, 'cat(x, y, z)' is equivalent to 'xyz.' Arguments **$4** through **$9** are null since no corresponding arguments were provided. Leading unquoted blanks, tabs, or newlines that occur during argument collection are discarded. All other white space is retained. Thus:

```
define(a, b c)
```

defines 'a' to be 'b c'.

Arguments are separated by commas; however, when commas are within parentheses, the argument is not terminated nor separated. For example:

```
define(a, (b,c))
```

has only two arguments. The first argument is **a**. The second is literally **(b,c)**. A bare comma or parenthesis can be inserted by quoting it.

Arithmetic Functions

M4 provides three built-in functions for doing arithmetic on integers (only). The simplest is **incr**, which increments its numeric argument by 1. The function **decr** decrements by 1. Thus, to handle the common programming situation where a variable is to be defined as "one more than N," use the following:

```
define(N, 100)
define(N1, 'incr(N)')
```

Then N1 is defined as one more than the current value of N.

The more general mechanism for arithmetic is a function called **eval**, which is capable of arbitrary arithmetic on integers. The operators in decreasing order of precedence are:

Precedent	Operator
Higher	unary + and -
	** (exponentiation)
	* / % (modulus)
	+ -
	== != < <= > >=
	! ~ (logical not and bitwise not)
	& (bitwise and)
	^ (bitwise or and exclusive or)
	&& (logical and)
Lower	\|\| (logical or)

Parentheses may be used to group operations where needed. All the operands of an expression given to **eval** must ultimately be numeric. The numeric value of a true relation (like 1>0) is 1 and false is 0. The precision in **eval** is 32 bits under the HP-UX operating system.

As a simple example, define M to be "2==N+1" using **eval** as follows:

```
define(N, 3)
define(M, 'eval(2==N+1)')
```

The defining text for a macro should be quoted unless the text is very simple. Quoting the defining text usually gives the desired result and is a good habit to get into.

File Manipulation

A new file can be included in the input at any time by the built-in function **include**. For example:

```
include(filename)
```

inserts the contents of *filename* in place of the **include** command. The contents of the file is often a set of definitions. The value of **include** (**include**'s replacement text) is the contents of the file. If needed, the contents can be captured in definitions, etc.

A fatal error occurs if the file named in **include** cannot be accessed. To get some control over this situation, the alternate form **sinclude** can be used. The function **sinclude** (silent include) says nothing and continues if the file named cannot be accessed.

The output of *m4* can be diverted to temporary files during processing, and the collected material can be output upon command. *M4* maintains nine of these diversions, numbers **1** through 9. If the built-in macro:

```
divert(n)
```

is used, all subsequent output is put onto the end of a temporary file referred to as *n*. Diverting to this file is stopped by the **divert** or **divert(0)** command, which resumes the normal output process.

Diverted text is normally output all at once at the end of processing with the diversions output in numerical order. Diversions can be brought back at any time by appending the new diversion to the current diversion. Output diverted to a stream other than 0 through 9 is discarded. The function **undivert** brings back all diversions in numerical order. The function **undivert** with arguments brings back the selected diversions in the order given. If the current diversion is not 0 through 9, the act of undiverting will discard text in the specified stream(s).

The value of **undivert** is *not* the diverted text. Furthermore, the diverted material is *not* re-scanned for macros. The function **divnum** returns the number of the currently active diversion. The current output stream is zero during normal processing.

System Command

Any program in the local operating system can be run by using the **syscmd** function. For example:

```
syscmd(date)
```

on the HP-UX system runs the **date** command. Normally, **syscmd** would be used to create a file for a subsequent **include**. To facilitate making unique file names, the function **maketemp** is provided with specifications identical to the system function *mktemp*. The **maketemp** macro replaces the string `XXXXXX` (6 `X`'s) anywhere in the argument with the process ID of the current process.

Conditionals

Arbitrary conditional testing is performed via the **ifelse** function. In the simplest form:

```
ifelse(a, b, c, d)
```

compares the two strings *a* and *b*. If *a* and *b* are identical, **ifelse** returns the string *c*. Otherwise, string *d* is returned. Thus, a macro called **compare** can be defined as one which compares two strings and returns "yes" or "no" if they are the same or different as follows:

```
define(compare, 'ifelse($1, $2, yes, no)')
```

Note the quotes, which prevents evaluation of **ifelse** from occurring too early. If the fourth argument is missing, it is treated as empty.

The function **ifelse** can actually have any number of arguments and provides a limited form of multi-way decision capability. In the input:

```
ifelse(a, b, c, d, e, f, g)
```

if the string *a* matches the string *b*, the result is *c*. Otherwise, if *d* is the same as *e*, the result is *f*. Otherwise, the result is *g*. If the final argument is omitted, the result is null, so:

```
ifelse(a, b, c)
```

is *c* if *a* matches *b*, and null otherwise.

String Manipulation

The **len** function returns the length of the string (number of characters) that makes up its argument. Thus:

```
len(abcdef)
```

is 6, and **len((a,b))** is 5.

The function **substr** can be used to produce substrings of strings. Using input,
substr(s, i, n) returns the substring of *s* that starts as the *i*th position (origin zero) and is *n* characters long. If *n* is omitted, the rest of the string is returned. The call:

```
substr('now is the time',1)
```

returns the following string:

```
ow is the time
```

If *i* or *n* is out of range, various actions occur.

The function **index(s1, s2)** returns the index (position) in *s1* where the string *s2* occurs or -1 if it does not occur. As with **substr**, the origin for strings is 0.

The function **translit** performs character transliteration and has the general form:

```
translit(s, f, t)
```

which modifies *s* by replacing any character found in *f* by the corresponding character of *t*. Using input:

```
translit(s, aeiou, 12345)
```

replaces the vowels by the corresponding digits. If *t* is shorter than *f*, characters that do not have an entry in *t* are deleted. As a limiting case, if *t* is not present at all, characters from *f* are deleted from *s*. So:

```
translit(s, aeiou)
```

would delete vowels from *s*.

There is also a function called **dnl** that deletes all characters that follow it up to and including the next new line. The **dnl** macro is useful mainly for throwing away empty lines that otherwise tend to clutter up *m4* output. Using input:

```
define(N, 100)
define(M, 200)
define(L, 300)
```

results in a new line at the end of each line that is not part of the definition. So the new line is copied into the output where it may not be wanted. If the function **dnl** is added to each of these lines, the newlines will disappear. Another method of achieving the same results is to input:

```
divert(-1)
define(...)
...
divert.
```

Printing

The **errprint** function writes its arguments out on the standard error file. An example would be:

```
errprint('fatal error')
```

The function **dumpdef** is a debugging aid that dumps the current names and definitions of items named as arguments. If no arguments are given, then all current names and definitions are printed. Do not forget to quote the names.

Notes

Table of Contents

Yacc: Yet Another Compiler-Compiler

Introduction 2
1: Basic Specifications 5
2: Actions 8
3: Lexical Analysis 11
4: How the Parser Works 13
5: Ambiguity and Conflicts 18
6: Precedence 23
7: Error Handling 26
8: The Yacc Environment 28
9: Hints for Preparing Specifications 30
Input Style 30
Left Recursion 31
Lexical Tie-ins 32
Reserved Words 33
10: Advanced Topics 33
Simulating Error and Accept in Actions 33
Accessing Values in Enclosing Rules. 34
Support for Arbitrary Value Types 35
References 37
Appendix A: A Simple Example 38
Appendix B: Yacc Input Syntax 40
Appendix C: An Advanced Example 42
Appendix D: Old Features Supported but Not Encouraged 48

Yacc: Yet Another Compiler-Compiler

Computer program input generally has some structure; in fact, every computer program that does input can be thought of as defining an "input language" which it accepts. An input language may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, usual input facilities are limited, difficult to use, and often are lax about checking their inputs for validity.

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

The input subroutine produced by Yacc calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or in terms of higher level constructs such as names and numbers. The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification.

Yacc is written in portable C. The class of specifications accepted is a very general one: LALR(1) grammars with disambiguating rules.

In addition to compilers for C, APL, Pascal, RATFOR, etc., Yacc has also been used for less conventional languages, including a phototypesetter language, several desk calculator languages, a document retrieval system, and a Fortran debugging system.

Introduction

Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a *parser*, calls the user-supplied low-level input routine (the *"lexical analyzer"*) to pick up the basic items (called *tokens*) from the input stream. These tokens are organized according to the input structure rules, called *"grammar rules"*; when one of these rules has been recognized, then user code supplied for this rule, an *action*, is invoked; actions have the ability to return values and make use of the values of other actions.

Yacc is written in a portable dialect of C[1] and the actions, and output subroutine, are in C as well. Moreover, many of the syntactic conventions of Yacc follow C.

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be

```
date : month_name day ',' year ;
```

Here, *date*, *month_name*, *day*, and *year* represent structures of interest in the input process; presumably, *month_name*, *day*, and *year* are defined elsewhere. The comma "," is enclosed in single quotes; this implies that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule, and have no significance in controlling the input. Thus, with proper definitions, the input

```
July 4, 1776
```

might be matched by the above rule.

An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. For historical reasons, a structure recognized by the lexical analyzer is called a *"terminal symbol"*, while the structure recognized by the parser is called a *"nonterminal symbol"*. To avoid confusion, terminal symbols will usually be referred to as *tokens*.

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. For example, the rules

```
month_name  :  'J' 'a' 'n'   ;
month_name  :  'F' 'e' 'b'   ;

          . . .

month_name  :  'D' 'e' 'c'   ;
```

might be used in the above example. The lexical analyzer would only need to recognize individual letters, and *month_name* would be a nonterminal symbol. Such low-level rules tend to waste time and space, and may complicate the specification beyond Yacc's ability to deal with it. Usually, the lexical analyzer would recognize the month names, and return an indication that a *month_name* was seen; in this case, *month_name* would be a token.

Literal characters such as "," must also be passed through the lexical analyzer, and are also considered tokens.

Specification files are very flexible. It is realively easy to add to the above example the rule

```
date  :  month '/' day '/' year   ;
```

allowing

```
7 / 4 / 1776
```

as a synonym for

```
July 4, 1776
```

In most cases, this new rule could be "slipped in" to a working system with minimal effort, and little danger of disrupting existing input.

The input being read may not conform to the specifications. These input errors are detected as early as is theoretically possible with a left-to-right scan; thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data can usually be quickly found. Error handling, provided as part of the input specifications, permits the reentry of bad data, or the continuation of the input process after skipping over the bad data.

In some cases, Yacc fails to produce a parser when given a set of specifications. For example, the specifications may be self contradictory, or they may require a more powerful recognition mechanism than that available to Yacc. The former cases represent design errors; the latter cases can often be corrected by making the lexical analyzer more powerful, or by rewriting some of the grammar rules. While Yacc cannot handle all possible specifications, its power compares favorably with similar systems; moreover, the constructions which are difficult for Yacc to handle are also frequently difficult for human beings to handle. Some users have reported that the discipline of formulating valid Yacc specifications for their input revealed errors of conception or design early in the program development.

The theory underlying Yacc has been described elsewhere [2] [3] [4]. Yacc has been extensively used in numerous practical applications, including *lint* [5], the Portable C Compiler [6], and a system for typesetting mathematics [7].

The next several sections describe the basic process of preparing a Yacc specification; Section 1 describes the preparation of grammar rules, Section 2 the preparation of the user supplied actions associated with these rules, and Section 3 the preparation of lexical analyzers. Section 4 describes the operation of the parser. Section 5 discusses various reasons why Yacc may be unable to produce a parser from a specification, and what to do about it. Section 6 describes a simple mechanism for handling operator precedences in arithmetic expressions. Section 7 discusses error detection and recovery. Section 8 discusses the operating environment and special features of the parsers Yacc produces. Section 9 gives some suggestions which should improve the style and efficiency of the specifications. Section 10 discusses some advanced topics, while Appendix A provides a brief example, and Appendix B gives a summary of the Yacc input syntax. Appendix C gives an example using some of the more advanced features of Yacc, and, finally, Appendix D describes mechanisms and syntax no longer actively supported, but provided for historical continuity with older versions of Yacc.

1: Basic Specifications

Names refer to either tokens or nonterminal symbols. Yacc requires token names to be declared as such. In addition, for reasons discussed in Section 3, it is often desirable to include the lexical analyzer as part of the specification file; it may be useful to include other programs as well. Thus, every specification file consists of three sections: the *declarations*, *"(grammar) rules"*, and *programs*. The sections are separated by double percent "%%" marks. (The percent "%" is generally used in Yacc specifications as an escape character.)

In other words, a full specification file looks like

```
declarations
%%
rules
%%
programs
```

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also; thus, the smallest legal Yacc specification is

```
%%
rules
```

Blanks, tabs, and newlines are ignored except that they may not appear in names or multi-character reserved symbols. Comments may appear wherever a name is legal; they are enclosed in /* . . . */, as in C.

The rules section is made up of one or more grammar rules. A grammar rule has the form:

```
A  :  BODY  ;
```

A represents a nonterminal name, and BODY represents a sequence of zero or more names and literals. The colon and the semicolon are Yacc punctuation.

Names may be of arbitrary length, and may be made up of letters, dot ".", underscore "_", and non-initial digits. Upper and lower case letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes. As in C, the backslash "\" is an escape character within literals, and all the C escapes are recognized. Thus

```
'\n'    newline
'\r'    return
'' single quote ''''
'\\'    backslash ''\''
'\t'    tab
'\b'    backspace
'\f'    form feed
'\xxx'  ''xxx'' in octal
```

For a number of technical reasons, the NUL character ('\0' or 0) should never be used in grammar rules.

If there are several grammar rules with the same left hand side, the vertical bar "|" can be used to avoid rewriting the left hand side. In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Thus the grammar rules

```
A       :       B  C  D   ;
A       :       E  F   ;
A       :       G   ;
```

can be given to Yacc as

```
A       :       B  C  D
        |       E  F
        |       G
;
```

It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input much more readable, and easier to change.

If a nonterminal symbol matches the empty string, this can be indicated in the obvious way:

```
empty :    ;
```

Names representing tokens must be declared; this is most simply done by writing

```
%token   name1  name2 . . .
```

in the declarations section. (See Sections 3 , 5, and 6 for much more discussion). Every name not defined in the declarations section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, one, called the *start symbol*, has particular importance. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left hand side of the first grammar rule in the rules section. It is possible, and in fact desirable, to declare the start symbol explicitly in the declarations section using the %start keyword:

```
%start   symbol
```

The end of the input to the parser is signaled by a special token, called the *endmarker*. If the tokens up to, but not including, the endmarker form a structure which matches the start symbol, the parser function returns to its caller after the endmarker is seen; it *accepts* the input. If the endmarker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the endmarker when appropriate; see section 3, below. Usually the endmarker represents some reasonably obvious I/O status, such as "end-of-file" or "end-of-record".

2: Actions

With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

An action is an arbitrary C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces "{" and "}". For example,

```
A       :       '(' B ')'
                    {       hello( 1, "abc" );  }
```

and

```
XXX     :       YYY ZZZ
                    {       printf("a message\n");
                            flag = 25;  }
```

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The symbol "dollar sign" "$" is used as a signal to Yacc in this context.

To return a value, the action normally sets the pseudo-variable "$$" to some value. For example, an action that does nothing but return the value 1 is

```
{ $$ = 1; }
```

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables $1, $2, . . ., which refer to the values returned by the components of the right side of a rule, reading from left to right. Thus, if the rule is

```
A       :       B  C  D   ;
```

for example, then $2 has the value returned by C, and $3 the value returned by D.

As a more concrete example, consider the rule

```
expr    :       '(' expr ')'   ;
```

The value returned by this rule is usually the value of the *expr* in parentheses. This can be indicated by

```
expr    :       '('  expr  ')'          {  $$ = $2 ;  }
```

By default, the value of a rule is the value of the first element in it ($1). Thus, grammar rules of the form

```
A       :       B    ;
```

frequently need not have an explicit action. This last rule is equivalent to

```
A:      B
          { $$ = $1; }
```

In the examples above, all the actions came at the end of their rules. Sometimes, it is desirable to get control before a rule is fully parsed. Yacc permits an action to be written in the middle of a rule as well as at the end. This rule is assumed to return a value, accessible through the usual mechanism by the actions to the right of it. In turn, it may access the values returned by the symbols to its left. Thus, in the rule

```
A       :       B
                        {  $$ = 1;  }
                C
                        {   x = $2;    y = $3;  }
        ;
```

the effect is to set *x* to 1, and *y* to the value returned by C.

Actions that do not terminate a rule are actually handled by Yacc by manufacturing a new nonterminal symbol name, and a new rule matching this name to the empty string. The interior action is the action triggered off by recognizing this added rule. Yacc actually treats the above example as if it had been written:

```
$ACT    :       /* empty */
                        {  $$ = 1;  }
        ;

A       :       B  $ACT  C
                        {   x = $2;    y = $3;  }
        ;
```

A good understanding of how Yacc handles interior actions can be important when interpreting conflict messages for such rules (see Section 5 of this tutorial). For example, conflicts in the grammar specification occur when an interior action occurs in a rule before the parser can be sure which rule is being reduced.

In many applications, output is not done directly by the actions; rather, a data structure, such as a parse tree, is constructed in memory, and transformations are applied to it before output is generated. Parse trees are particularly easy to construct, given routines to build and maintain the tree structure desired. For example, suppose there is a C function *node*, written so that the call

```
node( L, n1, n2 )
```

creates a node with label L, and descendants n1 and n2, and returns the index of the newly created node. Then parse tree can be built by supplying actions such as:

```
expr    :       expr  '+'  expr
                          { $$ = node('+', $1, $3); }
```

in the specification.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks "%{" and "%}". These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

```
%{   int variable = 0;   %}
```

could be placed in the declarations section, making *variable* accessible to all of the actions. The Yacc parser uses only names beginning in "yy" for its own internal variables, so users should avoid such names.

In these examples, all the values are integers: Section 10 discusses values of other types.

3: Lexical Analysis

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called *yylex*. The function returns an integer, the *"token number"*, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable *yylval*.

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by Yacc, or chosen by the user. In either case, the "# define" mechanism of C is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name DIGIT has been defined in the declarations section of the Yacc specification file. The relevant portion of the lexical analyzer might look like:

```
yylex(){
        extern int yylval;
        int c;
        . . .
        c = getchar();
        . . .
        switch( c ) {
                        . . .
        case '0':
        case '1':
          . . .
        case '9':
                yylval = c-'0';
                return( DIGIT );
                . . .
                }
        . . .
```

The intent is to return a token number of DIGIT, and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the programs section of the specification file, the identifier DIGIT will be defined as the token number associated with the token DIGIT.

This mechanism leads to clear, easily modified lexical analyzers; the only pitfall is the need to avoid using any token names in the grammar that are reserved or significant in C or the parser; for example, the use of token names *if* or *while* will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name *error* is reserved for error handling, and should not be used naively (see Section 7).

As mentioned above, the token numbers may be chosen by Yacc or by the user. In the default situation, the numbers are chosen by Yacc. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257.

To assign a token number to a token (including literals), the first appearance of the token name or literal *in the declarations section* can be immediately followed by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined by this mechanism retain their default definition. It is important that all token numbers be distinct.

For historical reasons, the endmarker must have token number 0 or negative. This token number cannot be redefined by the user; thus, all lexical analyzers should be prepared to return 0 or negative as a token number upon reaching the end of their input.

The *Lex* program is a very useful tool for constructing lexical analyzers. Lexical analyzers are designed to work in close harmony with Yacc parsers, but they use regular expressions instead of grammar rules. *Lex* can be easily used to produce quite complicated lexical analyzers, but there remain some languages (such as FORTRAN) which do not fit any theoretical framework, and whose lexical analyzers must be crafted by hand.

4: How the Parser Works

Yacc turns the specification file into a C program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex, and will not be discussed here (see the references for more information). The parser itself, however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by Yacc consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the *lookahead* token). The *"current state"* is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.

The machine has only four actions available to it, called *shift*, *reduce*, *accept*, and *error*. A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls *yylex* to obtain the next token.
2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off of the stack, and in the lookahead token being processed or left alone.

The *shift* action is the most common action the parser takes. Whenever a shift action is taken, there is always a lookahead token. For example, in state 56 there may be an action:

```
IF          shift 34
```

which says, in state 56, if the lookahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The lookahead token is cleared.

The *reduce* action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right hand side of a grammar rule, and is prepared to announce that it has seen an instance of the rule, replacing the right hand side by the left hand side. It may be necessary to consult the lookahead token to decide whether to reduce, but usually it is not; in fact, the default action (represented by a ".") is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, leading to some confusion. The action

```
reduce 18
```

refers to *grammar rule* 18, while the action

```
IF      shift 34
```

refers to *state* 34.

Suppose the rule being reduced is

```
A       : x  y  z   ;
```

The reduce action depends on the left hand symbol (A in this case), and the number of symbols on the right hand side (three in this case). To reduce, first pop off the top three states from the stack (In general, the number of states popped equals the number of symbols on the right side of the rule). In effect, these states were the ones put on the stack while recognizing *x*, *y*, and *z*, and no longer serve any useful purpose. After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule. Using this uncovered state, and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues. There are significant differences between the processing of the left hand symbol and an ordinary shift of a token, however, so this action is called a *goto* action. In particular, the lookahead token is cleared by a shift, and is not affected by a goto. In any case, the uncovered state contains an entry such as:

```
A       goto 20
```

causing state 20 to be pushed onto the stack, and become the current state.

In effect, the reduce action "turns back the clock" in the parse, popping the states off the stack to go back to the state where the right hand side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right hand side of the rule is empty, no states are popped off of the stack: the uncovered state is in fact the current state.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack, running in parallel with it, holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable *yylval* is copied onto the value stack. After the return from the user code, the reduction is carried out. When the *goto* action is done, the external variable *yyval* is copied onto the value stack. The pseudo-variables $1, $2, etc., refer to the value stack.

The other two parser actions are conceptually much simpler. The *accept* action indicates that the entire input has been seen and that it matches the specification. This action appears only when the lookahead token is the endmarker, and indicates that the parser has successfully done its job. The *error* action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen, together with the lookahead token, cannot be followed by anything that would result in a legal input. The parser reports an error, and attempts to recover the situation and resume parsing: the error recovery (as opposed to the detection of error) will be covered in Section 7.

It is time for an example! Consider the specification

```
%token  DING  DONG  DELL
%%
rhyme   :       sound  place
        ;
sound   :       DING  DONG
        ;
place   :       DELL
        ;
```

When Yacc is invoked with the **-v** option, a file called *y.output* is produced, with a human-readable description of the parser. The *y.output* file corresponding to the above grammar (with some statistics stripped off the end) is:

```
state 0
        $accept  :  _rhyme  $end

        DING  shift 3
        .  error

        rhyme  goto 1
        sound  goto 2

state 1
        $accept  :   rhyme_$end

        $end  accept
        .  error

state 2
        rhyme  :   sound_place

        DELL  shift 5
        .  error

        place   goto 4

state 3
```

```
        sound   :   DING_DONG

        DONG  shift 6
        .  error

state 4
        rhyme   :   sound  place_     (1)

 .   reduce  1

state 5
        place   :   DELL_     (3)

        .   reduce  3

state 6
        sound   :   DING  DONG_     (2)

        .   reduce  2
```

Notice that, in addition to the actions for each state, there is a description of the parsing rules being processed in each state. The _ character is used to indicate what has been seen, and what is yet to come, in each rule. Suppose the input is

```
DING  DONG  DELL
```

It is instructive to follow the steps of the parser while processing this input.

Initially, the current state is state 0. The parser needs to refer to the input in order to decide between the actions available in state 0, so the first token, *DING*, is read, becoming the lookahead token. The action in state 0 on *DING* is "shift 3", so state 3 is pushed onto the stack, and the lookahead token is cleared. State 3 becomes the current state. The next token, *DONG*, is read, becoming the lookahead token. The action in state 3 on the token *DONG* is "shift 6", so state 6 is pushed onto the stack, and the lookahead is cleared. The stack now contains 0, 3, and 6. In state 6, without even consulting the lookahead, the parser reduces by rule 2.

```
sound   :   DING  DONG
```

This rule has two symbols on the right hand side, so two states, 6 and 3, are popped off of the stack, uncovering state 0. Consulting the description of state 0, looking for a goto on *sound*,

```
sound   goto 2
```

is obtained; thus state 2 is pushed onto the stack, becoming the current state.

In state 2, the next token, *DELL*, must be read. The action is "shift 5", so state 5 is pushed onto the stack, which now has 0, 2, and 5 on it, and the lookahead token is cleared. In state 5, the only action is to reduce by rule 3. This has one symbol on the right hand side, so one state, 5, is popped off, and state 2 is uncovered. The goto in state 2 on *place*, the left side of rule 3, is state 4. Now, the stack contains 0, 2, and 4. In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again. In state 0, there is a goto on *rhyme* causing the parser to enter state 1. In state 1, the input is read; the endmarker is obtained, indicated by "$end" in the *y.output* file. The action in state 1 when the endmarker is seen is to accept, successfully ending the parse.

The reader is urged to consider how the parser works when confronted with such incorrect strings as

```
DING DONG DONG,
DING DONG,
DING DONG DELL DELL,
```

etc. A few minutes spent with this and other simple examples will probably be repaid when problems arise in more complicated contexts.

5: Ambiguity and Conflicts

A set of grammar rules is *ambiguous* if there is some input string that can be structured in two or more different ways. For example, the grammar rule

```
expr        :     expr  '-'  expr
```

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if the input is

```
expr  -  expr  -  expr
```

the rule allows this input to be structured as either

```
(  expr  -  expr  )  -  expr
```

or as

```
expr  -  (  expr  -  expr  )
```

(The first is called *"left association"*, the second *"right association"*).

Yacc detects such ambiguities when it is attempting to build the parser. It is instructive to consider the problem that confronts the parser when it is given an input such as

```
expr  -  expr  -  expr
```

When the parser has read the second expr, the input that it has seen:

```
expr  -  expr
```

matches the right side of the grammar rule above. The parser could *reduce* the input by applying this rule; after applying the rule; the input is reduced to *expr* (the left side of the rule). The parser would then read the final part of the input:

```
-  expr
```

and again reduce. The effect of this is to take the left associative interpretation.

Alternatively, when the parser has seen

```
expr - expr
```

it could defer the immediate application of the rule, and continue reading the input until it had seen

```
expr - expr - expr
```

It could then apply the rule to the rightmost three symbols, reducing them to *expr* and leaving

```
expr - expr
```

Now the rule can be reduced once more; the effect is to take the right associative interpretation. Thus, having read

```
expr - expr
```

the parser can do two legal things, a shift or a reduction, and has no way of deciding between them. This is called a *"shift / reduce conflict"*. It may also happen that the parser has a choice of two legal reductions; this is called a *"reduce / reduce conflict"*. Note that there are never any "Shift/shift" conflicts.

When there are shift/reduce or reduce/reduce conflicts, Yacc still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing which choice to make in a given situation is called a *"disambiguating rule"*.

Yacc invokes two disambiguating rules by default:

1. In a shift/reduce conflict, the default is to do the shift.
2. In a reduce/reduce conflict, the default is to reduce by the *earlier* grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred whenever there is a choice, in favor of shifts. Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but reduce/reduce conflicts should be avoided whenever possible.

Conflicts may arise because of mistakes in input or logic, or because the grammar rules, while consistent, require a more complex parser than Yacc can construct. The use of actions within rules can also cause conflicts, if the action must be done before the parser can be sure which rule is being recognized. In these cases, the application of disambiguating rules is inappropriate, and leads to an incorrect parser. For this reason, Yacc always reports the number of shift/reduce and reduce/reduce conflicts resolved by Rule 1 and Rule 2. To obtain more information about rules conflicts encountered by Yacc, invoke *yacc* with the **-v** option, then scan the resulting *y.output* file.

In general, whenever it is possible to apply disambiguating rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read but there are no conflicts. For this reason, most previous parser generators have considered conflicts to be fatal errors. Our experience has suggested that this rewriting is somewhat unnatural, and produces slower parsers; thus, Yacc will produce parsers even in the presence of conflicts.

As an example of the power of disambiguating rules, consider a fragment from a programming language involving an "if-then-else" construction:

```
stat    :       IF  '('  cond  ')'  stat_
        |       IF  '('  cond  ')'  stat_ELSE  stat
        ;
```

In these rules, *IF* and *ELSE* are tokens, *cond* is a nonterminal symbol describing conditional (logical) expressions, and *stat* is a nonterminal symbol describing statements. The first rule will be called the *simple-if* rule, and the second the *if-else* rule.

These two rules form an ambiguous construction, since input of the form

```
IF  (  C1  )  IF  (  C2  )  S1  ELSE  S2
```

can be structured according to these rules in two ways:

```
IF  (  C1  )  {
              IF  (  C2  )  S1
              }
ELSE  S2
```

or

```
IF  (  C1  )  {
              IF  (  C2  )  S1
              ELSE  S2
              }
```

The second interpretation is the one given in most programming languages having this construct. Each *ELSE* is associated with the last preceding "un-*ELSE*'d" *IF*. In this example, consider the situation where the parser has seen

```
IF  (  C1  )  IF  (  C2  )  S1
```

and is looking at the *ELSE*. It can immediately reduce by the simple-if rule to get

```
IF  (  C1  )  stat
```

and then read the remaining input,

```
ELSE  S2
```

and reduce

```
IF  (  C1  )  stat  ELSE  S2
```

by the if-else rule. This leads to the first of the above groupings of the input.

On the other hand, the *ELSE* may be shifted, *S2* read, and then the right hand portion of

```
IF  (  C1  )  IF  (  C2  )  S1  ELSE  S2
```

can be reduced by the if-else rule to get

```
IF  (  C1  )  stat
```

which can be reduced by the simple-if rule. This leads to the second of the above groupings of the input, which is usually desired.

Once again the parser can do two valid things — there is a shift/reduce conflict. The application of disambiguating rule 1 tells the parser to shift in this case, which leads to the desired grouping.

This shift/reduce conflict arises only when there is a particular current input symbol, *ELSE*, and particular inputs already seen, such as

```
IF  (  C1  )  IF  (  C2  )  S1
```

In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the state of the parser.

The conflict messages of Yacc are best understood by examining the verbose (**-v**) option output file. For example, the output corresponding to the above conflict state might be:

```
23: shift/reduce conflict (shift 45, reduce 18) on ELSE

state 23

        stat  :  IF  (  cond  )  stat           (18)
        stat  :  IF  (  cond  )  stat  ELSE  stat

        ELSE     shift 45
                 reduce 18
```

The first line describes the conflict, giving the state and the input symbol. The ordinary state description follows, giving the grammar rules active in the state, and the parser actions. Recall that the underline marks the portion of the grammar rules which has been seen. Thus in the example, in state 23 the parser has seen input corresponding to

```
IF  (  cond  )  stat
```

and the two grammar rules shown are active at this time. The parser can do two possible things. If the input symbol is *ELSE*, it is possible to shift into state 45. State 45 will have, as part of its description, the line

```
stat  :  IF  (  cond  )  stat  ELSE_stat
```

since the *ELSE* will have been shifted in this state. Back in state 23, the alternative action, described by ".", is to be done if the input symbol is not mentioned explicitly in the above actions; thus, in this case, if the input symbol is not *ELSE*, the parser reduces by grammar rule 18:

```
stat  :  IF  '('  cond  ')'  stat
```

Once again, notice that the numbers following "shift" commands refer to other states, while the numbers following "reduce" commands refer to grammar rule numbers. In the *y.output* file, the rule numbers are printed after those rules which can be reduced. In most states, only one reduce action is possible in the state; the default command. If you encounter unexpected shift/reduce conflicts, look at the verbose output to decide whether the default actions are appropriate. In really tough cases, you might need to know more about the behavior and construction of the parser than can be covered here. In this case, one of the theoretical references[2,3,4] might be appropriate, or the services of a local expert might be needed.

6: Precedence

There is one common situation where the rules given above for resolving conflicts are not sufficient; this is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of *precedence* levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form

```
expr : expr OP expr
```

and

```
expr : UNARY expr
```

for all binary and unary operators desired. This creates a very ambiguous grammar, with many parsing conflicts. As disambiguating rules, the user specifies the precedence, or binding strength, of all the operators, and the associativity of the binary operators. This information is sufficient to allow Yacc to resolve the parsing conflicts in accordance with these rules, and construct a parser that realizes the desired precedences and associativities.

The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with a Yacc keyword: %left, %right, or %nonassoc, followed by a list of tokens. All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. Thus,

```
%left '+' '-'
%left '*' '/'
```

describes the precedence and associativity of the four arithmetic operators. Plus and minus are left associative, and have lower precedence than star and slash, which are also left associative. The keyword %right is used to describe right associative operators, and the keyword %nonassoc is used to describe operators, like the operator .LT. in Fortran, that may not associate with themselves; thus,

```
A .LT. B .LT. C
```

is illegal in Fortran, and such an operator would be described with the keyword %nonassoc in Yacc. As an example of the behavior of these declarations, the description

```
%right  '='
%left   '+'  '-'
%left   '*'  '/'

%%

expr    :       expr  '='  expr
        |       expr  '+'  expr
        |       expr  '-'  expr
        |       expr  '*'  expr
        |       expr  '/'  expr
        |       NAME
        ;
```

might be used to structure the input

```
a = b = c*d - e - f*g
```

as follows:

```
a = ( b = ( ((c*d)-e) - (f*g) ) )
```

When this mechanism is used, unary operators must, in general, be given a precedence. Sometimes a unary operator and a binary operator have the same symbolic representation, but different precedences. An example is unary and binary '—'; unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword, %prec, changes the precedence level associated with a particular grammar rule. %prec appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal. It causes the precedence of the grammar rule to become that of the following token name or literal. For example, to make unary minus have the same precedence as multiplication the rules might resemble:

```
%left   '+'  '-'
%left   '*'  '/'

%%

expr    :       expr  '+'  expr
        |       expr  '-'  expr
        |       expr  '*'  expr
        |       expr  '/'  expr
        |       '-'  expr       %prec  '*'
        |       NAME
        ;
```

A token declared by %left, %right, and %nonassoc need not be, but may be, declared by %token as well.

The precedences and associativities are used by Yacc to resolve parsing conflicts; they give rise to disambiguating rules. Formally, the rules work as follows:

1. The precedences and associativities are recorded for those tokens and literals that have them.
2. A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last token or literal in the body of the rule. If the %prec construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them. This occurs when the last token or literal in the body of the rule (or the token referenced by **%prec** following a rule) has no defined precedence or associativity.
3. When there is a reduce/reduce conflict, or there is a shift/reduce conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two disambiguating rules given at the beginning of the section are used, and the conflicts are reported.
4. If there is a shift/reduce conflict, and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociating implies error.

Conflicts resolved by precedence are not counted in the number of shift/reduce and reduce/reduce conflicts reported by Yacc. This means that mistakes in the specification of precedences may disguise errors in the input grammar; it is a good idea to be sparing with precedences, and use them in an essentially "cookbook" fashion, until some experience has been gained. The *y.output* file is very useful in deciding whether the parser is actually doing what was intended.

7: Error Handling

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, it may be necessary to reclaim parse tree storage, delete or alter symbol table entries, and, typically, set switches to avoid generating any further output.

It is seldom acceptable to stop all processing when an error is found; it is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser "restarted" after an error. A general class of algorithms to do this involves discarding a number of tokens from the input string, and attempting to adjust the parser so that input can continue.

To allow the user some control over this process, Yacc provides a simple, but reasonably general, feature. The token name "error" is reserved for error handling. This name can be used in grammar rules; in effect, it suggests places where errors are expected, and recovery might take place. The parser pops its stack until it enters a state where the token "error" is legal. It then behaves as if the token "error" were the current lookahead token, and performs the action encountered. The lookahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted.

As an example, a rule of the form

```
stat    :    error
```

would, in effect, mean that on a syntax error the parser would attempt to skip over the statement in which the error was seen. More precisely, the parser will scan ahead, looking for three tokens that might legally follow a statement, and start processing at the first of these; if the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement, and end up reporting a second error where there is in fact no error.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Error rules such as the above are very general, but difficult to control. Somewhat easier are rules such as

```
stat    :       error   ';'
```

Here, when there is an error, the parser attempts to skip over the statement, but will do so by skipping to the next ';'. All tokens after the error and before the next ';' cannot be shifted, and are discarded. When the ';' is seen, this rule will be reduced, and any "cleanup" action associated with it performed.

Another form of error rule arises in interactive applications, where it may be desirable to permit a line to be reentered after an error. A possible error rule might be

```
input   :       error   '\n'  {  printf( "Reenter last line: " );  }  input
                              {  $$  =  $4; }
```

There is one potential difficulty with this approach; the parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens, and gives no message; this is clearly unacceptable. For this reason, there is a mechanism that can be used to force the parser to believe that an error has been fully recovered from. The statement

```
yyerrok ;
```

in an action resets the parser to its normal mode. The last example is better written

```
input   :       error   '\n'
                        {       yyerrok;
                                printf( "Reenter last line: " );    }
                input
                        {       $$  =  $4;  }
        ;
```

As mentioned above, the token seen immediately after the "error" symbol is the input token at which the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous lookahead token must be cleared. The statement

```
yyclearin ;
```

in an action will have this effect. For example, suppose the action after error were to call some sophisticated resynchronization routine, supplied by the user, that attempted to advance the input to the beginning of the next valid statement. After this routine was called, the next token returned by yylex would presumably be the first token in a legal statement; the old, illegal token must be discarded, and the error state reset. This could be done by a rule like

```
stat    :       error
                        {       resynch();
                                yyerrok ;
                                yyclearin ;     }
        ;
```

These mechanisms are admittedly crude, but do allow for a simple, fairly effective recovery of the parser from many errors; moreover, the user can get control to deal with the error actions required by other portions of the program.

8: The Yacc Environment

When the user inputs a specification to Yacc, the output is a file of C programs, called *y.tab.c* on most systems (due to local file system conventions, the names may differ from installation to installation). The function produced by Yacc is called *yyparse*\E\c it is an integer valued function. When it is called, it in turn repeatedly calls *yylex*, the lexical analyzer supplied by the user (see Section 3) to obtain input tokens. Eventually, either an error is detected, in which case (if no error recovery is possible) *yyparse* returns the value 1, or the lexical analyzer returns the endmarker token and the parser accepts. In this case, *yyparse* returns the value 0.

The user must provide a certain amount of environment for this parser in order to obtain a working program. For example, as with every C program, a program called *main* must be defined, that eventually calls *yyparse*. In addition, a routine called *yyerror* prints a message when a syntax error is detected.

These two routines must be supplied in one form or another by the user. To ease the initial effort of using Yacc, a library has been provided with default versions of *main* and *yyerror*. The name of this library is system dependent; on HP-UX systems, the library is accessed by a **−ly** argument to the loader. To show the triviality of these default programs, the source is given below:

```
main(){
      return( yyparse() );
      }
```

and

```
# include <stdio.h>

yyerror(s) char *s; {
                    fprintf( stderr, "%s\n", s );
                    }
```

The argument to *yyerror* is a string containing an error message, usually the string "syntax error". The average application will want to do better than this. Ordinarily, the program should keep track of the input line number, and print it along with the message when a syntax error is detected. The external integer variable *yychar* contains the lookahead token number at the time the error was detected; this may be of some interest in giving better diagnostics. Since the *main* program is probably supplied by the user (to read arguments, etc.) the Yacc library is useful only in small projects, or in the earliest stages of larger ones.

The external integer variable *yydebug* is normally set to 0. If it is set to a nonzero value, the parser will output a verbose description of its actions, including a discussion of which input symbols have been read, and what the parser actions are. Depending on the operating environment, it may be possible to set this variable by using a debugging system.

9: Hints for Preparing Specifications

This section contains miscellaneous hints on preparing efficient, easy to change, and clear specifications. The individual subsections are more or less independent.

Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file. The following style hints owe much to Brian Kernighan.

1. Use all capital letters for token names, all lower case letters for nonterminal names. This rule comes under the heading of "knowing who to blame when things go wrong."

2. Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.

3. Put all rules with the same left hand side together. Put the left hand side in only once, and let all following rules begin with a vertical bar.

4. Put a semicolon only after the last rule with a given left hand side, and put the semicolon on a separate line. This allows new rules to be easily added.

5. Indent rule bodies by two tab stops, and action bodies by three tab stops.

The example in Appendix A is written following this style, as are the examples in the text of this paper (where space permits). The user must make up his own mind about these stylistic questions; the central problem, however, is to make the rules visible through the morass of action code.

Left Recursion

The algorithm used by the Yacc parser encourages so called "left recursive" grammar rules: rules of the form

```
name    :       name  rest_of_rule  ;
```

These rules frequently arise when writing specifications of sequences and lists:

```
list    :       item
        |       list  ','  item
        ;
```

and

```
seq     :       item
        |       seq  item
        ;
```

In each of these cases, the first rule will be reduced for the first item only, and the second rule will be reduced for the second and all succeeding items.

With right recursive rules, such as

```
seq     :       item
        |       item  seq
        ;
```

the parser would be a bit bigger, and the items would be seen, and reduced, from right to left. More seriously, an internal stack in the parser would be in danger of overflowing if a very long sequence were read. Thus, the user should use left recursion wherever reasonable. It is worth considering whether a sequence with zero elements has any meaning, and if so, consider writing the sequence specification with an empty rule:

```
seq     :       /* empty */ |  seq  item
        ;
```

Once again, the first rule would always be reduced exactly once, before the first item was read, and then the second rule would be reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts might arise if Yacc is asked to decide which empty sequence it has seen, when it hasn't seen enough to know!

Lexical Tie-ins

Some lexical decisions depend on context. For example, the lexical analyzer might want to delete blanks normally, but not within quoted strings. Or names might be entered into a symbol table in declarations, but not in expressions.

One way of handling this situation is to create a global flag that is examined by the lexical analyzer, and set by actions. For example, suppose a program consists of 0 or more declarations, followed by 0 or more statements. Consider:

```
%{
        int dflag;
%}
   ...  other declarations ...

%%

prog    :       decls  stats
        ;

decls   :       /* empty */
                        {       dflag = 1;  }
        |       decls  declaration
        ;

stats   :       /* empty */
                        {       dflag = 0;  }
        |       stats  statement
        ;

     ...  other rules ...
```

The flag *dflag* is now 0 when reading statements, and 1 when reading declarations, *except for the first token in the first statement.* This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. In many cases, this single token exception does not affect the lexical scan.

This kind of "backdoor" approach can be elaborated to a noxious degree. Nevertheless, it represents a way of doing some things that are difficult, if not impossible, to do otherwise.

Reserved Words

Some programming languages permit the user to use words like "if", which are normally reserved, as label or variable names, provided that such use does not conflict with the legal use of these names in the programming language. This is extremely hard to do in the framework of Yacc; it is difficult to pass information to the lexical analyzer telling it "this instance of 'if' is a keyword, and that instance is a variable". The user can make a stab at it, using the mechanism described in the last subsection, but it is difficult.

A number of ways of making this easier are under advisement. Until then, it is better that the keywords be *reserved*; that is, be forbidden for use as variable names. There are powerful stylistic reasons for preferring this, anyway.

10: Advanced Topics

This section discusses a number of advanced features of Yacc.

Simulating Error and Accept in Actions

The parsing actions of error and accept can be simulated in an action by use of macros YYACCEPT and YYERROR. YYACCEPT causes *yyparse* to return the value 0; YYERROR causes the parser to behave as if the current input symbol had been a syntax error; *yyerror* is called, and error recovery takes place. These mechanisms can be used to simulate parsers with multiple endmarkers or context-sensitive syntax checking.

Accessing Values in Enclosing Rules.

An action may refer to values returned by actions to the left of the current rule. The mechanism is simply the same as with ordinary actions, a dollar sign followed by a digit, but in this case the digit may be 0 or negative. Consider

```
sent    :       adj  noun  verb  adj  noun
              {  look at the sentence . . .  }
        ;

adj     :       THE     {       $$ = THE;  }
        |       YOUNG   {       $$ = YOUNG;  }
        . . .
        ;

noun    :       DOG
                        {       $$ = DOG;  }
        |       CRONE
                        {       if( $0 == YOUNG ){
                                                printf( "what?\n" );
                                                   }
                                $$ = CRONE;
                        }
        ;
        . . .
```

In the action following the word CRONE, a check is made that the preceding token shifted was not YOUNG. Obviously, this is only possible when a great deal is known about what might precede the symbol *noun* in the input. There is also a distinctly unstructured flavor about this. Nevertheless, at times this mechanism will save a great deal of trouble, especially when a few combinations are to be excluded from an otherwise regular structure.

Support for Arbitrary Value Types

By default, the values returned by actions and the lexical analyzer are integers. Yacc can also support values of other types, including structures. In addition, Yacc keeps track of the types, and inserts appropriate union member names so that the resulting parser will be strictly type checked. The Yacc value stack (see Section 4) is declared to be a *union* of the various types of values desired. The user declares the union, and associates union member names to each token and nonterminal symbol having a value. When the value is referenced through a $$ or $n construction, Yacc will automatically insert the appropriate union name, so that no unwanted conversions will take place. In addition, type checking commands such as *Lint*[5] will be far more silent.

There are three mechanisms used to provide for this typing. First, there is a way of defining the union; this must be done by the user since other programs, notably the lexical analyzer, must know about the union member names. Second, there is a way of associating a union member name with tokens and nonterminals. Finally, there is a mechanism for describing the type of those few values where Yacc can not easily determine the type.

To declare the union, the user includes in the declaration section:

```
%union  {
        body of union ...
        }
```

This declares the Yacc value stack, and the external variables *yylval* and *yyval*, to have type equal to this union. If Yacc was invoked with the **−d** option, the union declaration is copied onto the *y.tab.h* file. Alternatively, the union may be declared in a header file, and a typedef used to define the variable YYSTYPE to represent this union. Thus, the header file might also have said:

```
typedef union {
        body of union ...
        } YYSTYPE;
```

The header file must then also be included in the declarations section, by use of %{ and %}.

Once YYSTYPE is defined, the union member names must be associated with the various terminal and nonterminal names. The construction

```
< name >
```

is used to indicate a union member name. If this follows one of the keywords %token, %left, %right, and %nonassoc, the union member name is associated with the tokens listed. Thus, saying

```
%left  <optype>  '+'  '-'
```

will cause any reference to values returned by these two tokens to be tagged with the union member name *optype*. Another keyword, %type, is used similarly to associate union member names with nonterminals. Thus, one might say

```
%type  <nodetype>  expr  stat
```

There remain a couple of cases where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no *"a priori"* type. Similarly, reference to left context values (such as $0 – see the previous subsection) leaves Yacc with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name, between < and >, immediately after the first $. An example of this usage is

```
rule    :       aaa  {  $<intval>$  =  3;  } bbb
                     {       fun( $<intval>2, $<other>0 );  }
        ;
```

This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in Appendix C. The facilities in this subsection are not triggered until they are used: in particular, the use of %type will turn on these mechanisms. When they are used, there is a fairly strict level of checking. For example, use of $n or $$ to refer to something with no defined type is diagnosed. If these facilities are not triggered, the Yacc value stack is used to hold *int*'s, as was true historically.

References

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey (1978).

2. A. V. Aho and S. C. Johnson, "LR Parsing," *Comp. Surveys* **6**(2) pp. 99-124 (June 1974).

3. A. V. Aho, S. C. Johnson, and J. D. Ullman, "Deterministic Parsing of Ambiguous Grammars," *Comm. Assoc. Comp. Mach.* **18**(8) pp. 441-452 (August 1975).

4. A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, Mass. (1977).

5. S. C. Johnson, "Lint, a C Program Checker," Comp. Sci. Tech. Rep. No. 65 (December 1977).

6. S. C. Johnson, "A Portable Compiler: Theory and Practice," *Proc. 5th ACM Symp. on Principles of Programming Languages*, (January 1978).

7. B. W. Kernighan and L. L. Charry, "A System for Typesetting Mathematics," *Comm. Assoc. Comp. Mach.* **18** pp. 151-157 (March 1975).

8. M. E. Lesk, "Lex – A Lexical Analyzer Generator," Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey (October 1975). (See *HP-UX Concepts and Tutorials*, Vol. 1.)

Appendix A: A Simple Example

This example gives the complete Yacc specification for a small desk calculator; the desk calculator has 26 registers, labeled "a" through "z", and accepts arithmetic expressions made up of the operators +, −, *, /, % (mod operator), & (bitwise and), | (bitwise or), and assignment. If an expression at the top level is an assignment, the value is not printed; otherwise it is. As in C, an integer that begins with 0 (zero) is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a Yacc specification, the desk calculator does a reasonable job of showing how precedences and ambiguities are used, and demonstrating simple error recovery. The major oversimplifications are that the lexical analysis phase is much simpler than for most applications, and the output is produced immediately, line by line. Note the way that decimal and octal integers are read in by the grammar rules; This job is probably better done by the lexical analyzer.

```
%{
# include <stdio.h>
# include <ctype.h>

int regs[26];
int base;

%}

%start list

%token DIGIT LETTER

%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS      /* supplies precedence for unary minus */

%%      /* beginning of rules section */

list    :       /* empty */
        |       list stat '\n'
        |       list error '\n'
                        {       yyerrok; }
        ;

stat    :       expr
                        {       printf( "%d\n", $1 ); }
        |       LETTER '=' expr
                        {       regs[$1] = $3; }
        ;
```

```
expr    :       '('  expr  ')'
                        {       $$  =  $2;  }
        |       expr  '+'  expr
                        {       $$  =  $1  +  $3;  }
        |       expr  '-'  expr
                        {       $$  =  $1  -  $3;  }
        |       expr  '*'  expr
                        {       $$  =  $1  *  $3;  }
        |       expr  '/'  expr
                        {       $$  =  $1  /  $3;  }
        |       expr  '%'  expr
                        {       $$  =  $1  %  $3;  }
        |       expr  '&'  expr
                        {       $$  =  $1  &  $3;  }
        |       expr  '|'  expr
                        {       $$  =  $1  |  $3;  }
        |       '-'  expr       %prec  UMINUS
                        {       $$  =  -  $2;  }
        |       LETTER
                        {       $$  =  regs[$1];  }
        |       number
        ;

number  :       DIGIT
                        {       $$ = $1;    base  =  ($1==0)  ?  8  :  10;  }
        |       number  DIGIT
                        {       $$  =  base * $1  +  $2;  }
        ;

%%      /*  start  of  programs  */

yylex() {               /*  lexical  analysis  routine  */
        /*  returns  LETTER  for  a  lower  case  letter,  yylval = 0
through 25  */
        /*  return  DIGIT  for  a  digit,  yylval = 0  through  9  */
        /*  all  other  characters  are  returned  immediately  */

        int  c;

        while(  (c=getchar())  ==  ' '  )  {    /*  skip  blanks  */  }

        /*  c  is  now  nonblank  */

        if(  islower(  c  )  )  {
                                yylval  =  c  -  'a';
                                return  (  LETTER  );
                                }
        if(  isdigit(  c  )  )  {
                                yylval  =  c  -  '0';
                                return(  DIGIT  );
```

```
                                }
        return(  c  );
        }
```

Appendix B: Yacc Input Syntax

This Appendix has a description of the Yacc input syntax, as a Yacc specification. Context dependencies, etc., are not considered. Ironically, the Yacc input specification language is most naturally specified as an LR(2) grammar; the sticky part comes when an identifier is seen in a rule, immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise it is a continuation of the current rule, which just happens to have an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier, and decide whether the next token (skipping blanks, newlines, comments, etc.) is a colon. If so, it returns the token C_IDENTIFIER. Otherwise, it returns IDENTIFIER. Literals (quoted strings) are also returned as IDENTIFIERS, but never as part of C_IDENTIFIERs.

```
            /*  grammar  for  the  input  to  Yacc  */

        /*  basic  entities  */
   %token  IDENTIFIER      /*   includes  identifiers   and  literals  */
   %token  C_IDENTIFIER    /*    identifier  (but  not  literal)  followed  by
colon    */
   %token  NUMBER          /*    [0-9]+    */

        /*  reserved  words:    %type  =>  TYPE,  %left  =>  LEFT,  etc.  */

   %token  LEFT  RIGHT  NONASSOC  TOKEN  PREC  TYPE  START  UNION

   %token  MARK    /*  the  %%  mark  */
   %token  LCURL   /*  the  %{  mark  */
   %token  RCURL   /*  the  %}  mark  */

        /*  ascii  character  literals  stand  for  themselves  */

   %start  spec

   %%

   spec    :       defs  MARK  rules  tail
           ;

   tail    :       MARK    {    In this action, eat up the rest of the file  }
           |       /*  empty:  the  second  MARK  is  optional  */
           ;
```

```
defs    :       /* empty */
        |       defs def
        ;

def     :       START IDENTIFIER
        |       UNION { Copy union definition to output }
        |       LCURL { Copy C code to output file } RCURL
        |       ndefs rword tag nlist
        ;

rword   :       TOKEN
        |       LEFT
        |       RIGHT
        |       NONASSOC
        |       TYPE
        ;

tag     :       /* empty: union tag is optional */
        |       '<' IDENTIFIER '>'
        ;

nlist   :       nmno
        |       nlist nmno
        |       nlist ',' nmno
        ;

nmno    :       IDENTIFIER              /* NOTE: literal illegal with
%type */
        |       IDENTIFIER NUMBER       /* NOTE: illegal with %type */
        ;

       /* rules section */

rules   :       C_IDENTIFIER rbody prec
        |       rules rule
        ;

rule    :       C_IDENTIFIER rbody prec
        |       '|' rbody prec
        ;

rbody   :       /* empty */
        |       rbody IDENTIFIER
        |       rbody act
        ;

act     :       '{' { Copy action, translate $$, etc. } '}'
    ;
```

```
prec    :       /* empty */
        |       PREC IDENTIFIER
        |       PREC IDENTIFIER act
        |       prec ';'
        ;
```

Appendix C: An Advanced Example

This Appendix gives an example of a grammar using some of the advanced features discussed in Section 10. The desk calculator example in Appendix A is modified to provide a desk calculator that does floating point interval arithmetic. The calculator understands floating point constants, the arithmetic operations +, −, *, /, unary −, and = (assignment), and has 26 floating point variables, "a" through "z". Moreover, it also understands *intervals*, written

```
( x , y )
```

where x is less than or equal to y. There are 26 interval valued variables "A" through "Z" that may also be used. The usage is similar to that in Appendix A; assignments return no value, and print nothing, while expressions print the (floating or interval) value.

This example explores a number of interesting features of Yacc and C. Intervals are represented by a structure, consisting of the left and right endpoint values, stored as *double*'s. This structure is given a type name, INTERVAL, by using *typedef*. The Yacc value stack can also contain floating point scalars, and integers (used to index into the arrays holding the variable values). Notice that this entire strategy depends strongly on being able to assign structures and unions in C. In fact, many of the actions call functions that return structures as well.

It is also worth noting the use of YYERROR to handle error conditions: division by an interval containing 0, and an interval presented in the wrong order. In effect, the error recovery mechanism of Yacc is used to throw away the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (e.g. scalar or interval) of intermediate expressions. Note that a scalar can be automatically promoted to an interval if the context demands an interval value. This causes a large number of conflicts when the grammar is run through Yacc: 18 Shift/Reduce and 26 Reduce/Reduce. The problem can be seen by looking at the two input lines:

```
2.5 + ( 3.5 - 4. )
```

and

```
2.5 + ( 3.5 , 4. )
```

Notice that the 2.5 is to be used in an interval valued expression in the second example, but this fact is not known until the "," is read; by this time, 2.5 is finished, and the parser cannot go back and change its mind. More generally, it might be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. This problem is evaded by having two rules for each binary interval valued operator: one when the left operand is a scalar, and one when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion will be applied automatically. Despite this evasion, there are still many cases where the conversion may be applied or not, leading to the above conflicts. They are resolved by listing the rules that yield scalars first in the specification file; in this way, the conflicts will be resolved in the direction of keeping scalar valued expressions scalar valued until they are forced to become intervals.

This way of handling multiple types is very instructive, but not very general. If there were many kinds of expression types, instead of just two, the number of rules needed would increase dramatically, and the conflicts even more dramatically. Thus, while this example is instructive, it is better practice in a more normal programming language environment to keep the type information as part of the value, and not as part of the grammar.

Finally, a word about the lexical analysis. The only unusual feature is the treatment of floating point constants. The C library routine *atof* is used to do the actual conversion from a character string to a double precision value. If the lexical analyzer detects an error, it responds by returning a token that is illegal in the grammar, provoking a syntax error in the parser, and thence error recovery.

```
%{

# include <stdio.h>
# include <ctype.h>

typedef struct interval {
                        double lo, hi;
                        } INTERVAL;

INTERVAL vmul(), vdiv();

double atof();

double dreg[ 26 ];
INTERVAL vreg[ 26 ];

%}

%start    lines

%union    {
          int ival;
          double dval;
          INTERVAL vval;
          }

%token <ival> DREG VREG      /* indices into dreg, vreg arrays */

%token <dval> CONST          /* floating point constant */

%type <dval> dexp            /* expression */

%type <vval> vexp            /* interval expression */

        /* precedence information about the operators */

%left   '+' '-'
%left   '*' '/'
%left   UMINUS       /* precedence for unary minus */

%%

lines   :       /* empty */
        |       lines line
        ;

line    :       dexp '\n'
                        {       printf( "%15.8f\n", $1 ); }
```

```
        |       vexp  '\n'
                        {       printf( "(%15.8f, %15.8f)\n", $1.
lo, $1.hi ); }
        |       DREG  '='  dexp  '\n'
                        {       dreg[$1]  =  $3;  }
        |       VREG  '='  vexp  '\n'
                        {       vreg[$1]  =  $3;  }
        |       error  '\n'
                        {       yyerrok;  }
        ;

dexp    :       CONST
        |       DREG
                        {       $$  =  dreg[$1];  }
        |       dexp  '+'  dexp
                        {       $$  =  $1  +  $3;  }
        |       dexp  '-'  dexp
                        {       $$  =  $1  -  $3;  }
        |       dexp  '*'  dexp
                        {       $$  =  $1  *  $3;  }
        |       dexp  '/'  dexp
                        {       $$  =  $1  /  $3;  }
        |       '-'  dexp       %prec  UMINUS
                        {       $$  =  - $2;  }
        |       '('  dexp  ')'
                        {       $$  =  $2;  }
        ;

vexp    :       dexp
                        {       $$.hi  =  $$.lo  =  $1;  }
        |       '('  dexp  ','  dexp  ')'
                        {
                        $$.lo  =  $2;
                        $$.hi  =  $4;
                        if(  $$.lo > $$.hi  ){
                                        printf("interval out of order
\n");
                                        YYERROR;
                                        }
                        }
        |       VREG
                        {       $$  =  vreg[$1];    }
        |       vexp  '+'  vexp
                        {       $$.hi  =  $1.hi  +  $3.hi;
                                $$.lo  =  $1.lo  +  $3.lo;    }
        |       dexp  '+'  vexp
                        {       $$.hi  =  $1  +  $3.hi;
                                $$.lo  =  $1  +  $3.lo;    }
        |       vexp  '-'  vexp
                        {       $$.hi  =  $1.hi  -  $3.lo;
```

```
                        $$.lo = $1.lo - $3.hi;    }
        |       dexp '-' vexp
                  {     $$.hi = $1 - $3.lo;
                        $$.lo = $1 - $3.hi;    }
        |       vexp '*' vexp
                  {     $$ = vmul( $1.lo, $1.hi, $3 ); }
        |       dexp '*' vexp
                  {     $$ = vmul( $1, $1, $3 ); }
        |       vexp '/' vexp
                  {     if( dcheck( $3 ) ) YYERROR;
                        $$ = vdiv( $1.lo, $1.hi, $3 ); }
        |       dexp '/' vexp
                  {     if( dcheck( $3 ) ) YYERROR;
                        $$ = vdiv( $1, $1, $3 ); }
        |       '-' vexp        %prec UMINUS
                  {     $$.hi = -$2.lo;    $$.lo = -$2.hi;    }
        |       '(' vexp ')'
                  {     $$ = $2; }
        ;

%%

# define BSZ 50         /* buffer size for floating point numbers */

        /* lexical analysis */

yylex(){
        register c;

        while( (c=getchar()) == ' ' ){ /* skip over blanks */ }

        if( isupper( c ) ){
                        yylval.ival = c - 'A';
                        return( VREG );
                        }
        if( islower( c ) ){
                        yylval.ival = c - 'a';
                        return( DREG );
                        }

        if( isdigit( c ) || c=='.' ){
                /* gobble up digits, points, exponents */

                char buf[BSZ+1], *cp = buf;
                int dot = 0, exp = 0;

                for( ; (cp-buf)<BSZ ; ++cp,c=getchar() ){

                        *cp = c;
                        if( isdigit( c ) ) continue;
```

```
                        if( c == '.' ){
                                if( dot++ || exp ) return( '.' );
                                /* will cause syntax error */
                                continue;
                                }

                        if( c == 'e' ){
                                if( exp++ ) return( 'e' );
                                /* will cause syntax error */
                                continue;
                                }

                        /* end of number */
                        break;
                        }
                *cp = '\0';
                if((cp-buf) >= BSZ) printf("constant too long: truncated\n");
                else ungetc( c, stdin );  /* push back last char read
*/
                yylval.dval = atof( buf );
                return( CONST );
                }
        return( c );
        }

   INTERVAL hilo( a, b, c, d ) double a, b, c, d; {
        /* returns the smallest interval containing a, b, c, and
d */
        /* used by *, / routines */
        INTERVAL v;

        if( a>b ) { v.hi = a;   v.lo = b; }
        else { v.hi = b;   v.lo = a; }

        if( c>d ) {
                if( c>v.hi ) v.hi = c;
                if( d<v.lo ) v.lo = d;
                }
        else    {
                if( d>v.hi ) v.hi = d;
                if( c<v.lo ) v.lo = c;
                }
        return( v );
        }

   INTERVAL vmul( a, b, v ) double a, b;   INTERVAL v; {
        return( hilo( a*v.hi, a*v.lo, b*v.hi, b*v.lo ) );
        }

   dcheck( v ) INTERVAL v; {
```

```
        if( v.hi >= 0. && v.lo <= 0. ){
                printf( "divisor interval contains 0.\n" );
                return( 1 );
                }
        return( 0 );
        }

INTERVAL vdiv( a, b, v ) double a, b;   INTERVAL v; {
        return( hilo( a/v.hi, a/v.lo, b/v.hi, b/v.lo ) );
        }
```

Appendix D: Old Features Supported but Not Encouraged

This Appendix mentions synonyms and features which are supported for historical continuity, but, for various reasons, are not encouraged.

1. Literals may also be delimited by double quotes """.

2. Literals may be more than one character long. If all the characters are alphabetic, numeric, or _, the type number of the literal is defined, just as if the literal did not have the quotes around it. Otherwise, it is difficult to find the value for such literals.

 The use of multi-character literals is likely to mislead those unfamiliar with Yacc, since it suggests that Yacc is doing a job which must be actually done by the lexical analyzer.

3. Most places where % is legal, backslash "\" may be used. In particular, \\ is the same as %%, \left the same as %left, etc.

4. There are a number of other synonyms:

   ```
   %< is the same as %left
   %> is the same as %right
   %binary and %2 are the same as %nonassoc
   %0 and %term are the same as %token
   %= is the same as %prec
   ```

5. Actions may also have the form

   ```
   ={ . . . }
   ```

 and the curly braces can be dropped if the action is a single C statement.

6. C code between %{ and %} used to be permitted at the head of the rules section, as well as in the declaration section.

Index

a

accept action 15
accept and *error*, simulating in actions 33
accessing values enclosed in rules 34
action defined 8
actions, user-supplied 8–10
ambiguity 18–21
ambiguity; disambiguating rules 25
arithmetic operators 23, 24
association, left/right 18

b

binary operators 23

d

disambiguating rules 25

e

endmarker 7
environment, yacc 28–29
error and *accept*, simulating in actions 33
error detection, input 3
error handling 28, 29
error used as token name 11
example, advanced grammar 42
example yacc specification 38

g

grammar rules 2

h

handling shift actions 14

i

input syntax, yacc 40
interior actions, handling of 9

l

left-hand side of grammar rules repeated 6
left/right association 18
lexical analysis 11–12
lexical analyzer 2
literal 6
literal character, token number for 12
literal characters treated as tokens 3

n

NULL character not allowed in grammar rules 6

o

obsolete features supported 48

p

parser operation 13–17
parser rules processing described 16
precedence 23–25
preparation of grammar rules 5–7

r

reduce parser action 13
right/left association 18

s

shift parser action 13
simulating *accept* and *error* in actions 33
specification file structure 5
specifications:
 input style 30
 left recursion 31
 lexical tie-ins 32
 reserved words 33
start symbol 7
syntax, yacc input 40

t

token names declared 5
token number 11
token number for literal characters 12
tokens defined 2

u

unary operators 23, 24
unELSEd IF 20–21
user-supplied actions 8–10

v

-v option when yacc is invoked 15, 20
value types, arbitrary, support for 35
values enclosed in rules, accessing 34

y

yacc environment 28–29
yacc input syntax 40
yacc specification example 38

Table of Contents

The ADB Debugger

Introduction 1
Invocation 1
Command Format 2
Displaying Information 3
Debugging C Programs 6
Debugging a Core Image 6
Setting Breakpoints 9
Advanced Breakpoint Usage 14
Other Breakpoint Facilities 16
Maps 17
Variables and Registers 20
Formatted Dumps 21
Patching 25
Anomalies 26
Command Summary 27
Formatted Printing 27
Breakpoint and Program Control 27
Miscellaneous Printing 27
Calling the Shell 28
Assignment to Variables 28
Format Summary 28
Expression Summary 29
Expression Components 29
Dyadic Operators 29
Monadic Operators 29

The ADB Debugger

Introduction

ADB is a debugging program that is available on HP-UX. It provides capabilities to look at "core" files resulting from aborted programs, print output in a variety of formats, patch files, and run programs with embedded breakpoints. This document provides examples of the more useful features of ADB.

Invocation

ADB is invoked as:

```
adb objfile corefile
```

where `objfile` is an executable HP-UX file and `corefile` is a core image file. Many times this will look like:

```
adb a.out core
```

or more simply:

```
adb
```

where the defaults are `a.out` and `core` respectively. The filename minus (-) means "ignore this argument," as in:

```
adb - core
```

The `objfile` can be written to if **adb** is invoked with the `-w` flag as in:

```
adb -w a.out -
```

ADB catches signals, so a user cannot use a quit signal to exit from ADB. The request `$q` or `$Q` (or CTRL-D) must be used to exit from ADB.

Command Format

The general form of a request is:

[address] [,count] [command] [modifier]

ADB maintains a current address, called *dot*, similar in function to the current pointer in the HP-UX editor. When **address** is entered, dot is set to that location. The command is then executed count times.

Address and count are represented by expressions. Expressions are made up from decimal, octal, and hexadecimal integers, and symbols from the program under test. These may be combined with the operators +, -, *, % (integer division), & (bitwise and), | (bitwise inclusive or), # (round up to the next multiple), and ~ (not). (All arithmetic within ADB is 32 bits.) When typing a symbolic address for a C program, the user can type `name` or `_name`; ADB will recognize both forms. The default base for integer input is initialized to hexadecimal, but can be changed.

The following table illustrates some general ADB commands and meanings:

?	Print contents from `a.out` file
/	Print contents from `core` file
=	Print value of "dot"
:	Breakpoint control
$	Miscellaneous requests
;	Request separator
!	Escape to shell

CTRL-C terminates execution of any ADB command.

Displaying Information

ADB has requests for examining locations in either **objfile** or **corefile**. The ? request examines the contents of **objfile**, the / request examines the **corefile**.

Following the ? or / command the user specifies a format.

The following are some commonly used format letters.

`c`	one byte as a character
`x`	two bytes in hexadecimal
`X`	four bytes in hexadecimal
`d`	two bytes in decimal
`F`	eight bytes in double floating point
`i`	MC68000 instruction
`s`	a null terminated character string
`a`	print in symbolic form
`n`	print a newline
`r`	print a blank space
`^`	backup dot

A command to print the first hexadecimal element of an array of long integers named `ints` in C would look like:

```
ints/X
```

This instruction would set the value of **dot** to the symbol table value of `_ints`. It would also set the value of the dot increment to four. The dot increment is the number of bytes printed by the format.

Let us say that we wanted to print the first four bytes as a hexadecimal number and the next four as a decimal one. We could do this by:

```
ints/XD
```

In this case, **dot** would still be set to `_ints` and the dot increment would be eight bytes. The dot increment is the value which is used by the `newline` command. `Newline` is a special command which repeats the previous command. It does not always have meaning. In this context, it means to repeat the previous command using a count of one and an address of dot plus dot increment. In this case, `newline` would set dot to `ints+0x8` and type the two long integers it found there, the first in hex and the second in decimal. The `newline` command can be repeated as often as desired and this can be used to scroll through sections of memory.

Using the above example to illustrate another point, let us say that we wanted to print the first four bytes in long hex format and the next four bytes in byte hex format. We could do this by:

```
ints/X4b
```

Any format character can be preceded by a decimal repeat character.

The count field can be used to repeat the entire format as many times as desired. In order to print three lines using the above format we would type

```
ints,3/X4bn
```

The `n` on the end of the format is used to output a carriage return and make the output much easier to read.

In this case the value of dot will not be `_ints`. It will rather be `_ints+0x10`. Each time the format was re-executed dot would have been set to dot plus dot increment. Thus the value of dot would be the value that dot had at the beginning of the last execution of the format. Dot increment would be the size of the format: eight bytes. A `newline` command at this time would set. dot to `ints+0x18` and print only one repetition of the format, since the count would have been reset to one.

In order to see what the value of dot is at this point the command

```
.=a
```

could be typed. = is a command which can be used to print the value of **address** in any format. It is also possible to use this command to convert from one base to another:

```
0x32=oxd
```

This will print the value `0x32` in octal, hexadecimal and decimal.

Complicated formats are remembered by ADB. One format is remembered for each of the ? , / and = commands. This means that it is possible to type

```
0x64=
```

and have the value `0x64` printed out in octal, hex and decimal. And after that, type

```
ints/
```

and have ADB print out four bytes in long hex format and four bytes in byte hex format.

To an observant individual it might seem that the two commands

```
main,10?i
```

and

```
main?10i
```

would be the same.

There are two differences. The first is that the numbers are in a different base. The repeat factor can only be a decimal constant, while the count can be an expression and is therefore, by default, in a hex base.

The second difference is that a `newline` after the first command would print one line, while a `newline` after the second command would print another ten lines.

Debugging C Programs

The following examples illustrate various features of ADB. Certain parts of the output (such as machine addresses) may depend on the hardware being used, as well as how the program was linked (unshared, shared, or demand loaded).

Debugging a Core Image

Consider the C program in Figure 1. The program is used to illustrate some of the useful information that can be obtained from a core file. The object of the program is to calculate the square of the variable `ival` by calling the function `sqr` with the address of the integer. The error is that the value of the integer is being passed rather than the address of the integer. Executing the program produces a core file because of a bus error.

Figure 1: C program with pointer bug

```
int ints[]=    {1,2,3,4,5,6,7,8,9,0,
                1,2,3,4,5,6,7,8,9,0,
                1,2,3,4,5,6,7,8,9,0,
                1,2,3,4,5,6,7,8,9,0};
int ival;
main()
{
        register int i;
        for(i=0;i<10;i++)
        {       ival = ints[i];
                sqr(ival);
                printf("sqr of %d is %d\n",ints[i],ival);
        }
}

sqr(x)
int *x;
{
        *x *= *x;
}
```

ADB is invoked by:

```
adb
```

The first debugging request:

```
$c
```

is used to give a C backtrace through the subroutines called. This request can be used to check the validity of the parameters passed. As shown in Figure 2 we can see that the value passed on the stack to the routine `sqr` is a 1, which is not what we are expecting.

Figure 2: ADB output for program of Figure 1

```
$c
_main+0x2A:     _sqr    (0x1)
start+0x48:     _main   (0x1, 0xFFFF7DEC)
$r
ps      0x0
pc      0xFE    _sqr+0x3E:      unlk    a6

sp  0xFFFF7DC4

d0      0x1AE9          a0  0x1
d1      0x53            a1  0xFFFF7DEC
d2      0xFFC01         a2  0xFFC8B004
d3      0xFFC90405      a3  0x1F5BE
d4      0xFFC90401      a4  0x1F604
d5      0x700           a5  0x1F380
d6      0x0             a6  0xFFFF7DC8
d7      0x0             sp  0xFFFF7DC4
sqr+0x34,5?ia
_sqr+0x34:              move.w  (a7)+,d0
_sqr+0x36:              mulu    d1,d0
_sqr+0x38:              move.l  0x8(a6),(a0)
_sqr+0x3C:              move.l  d0,(a0)
_sqr+0x3E:              unlk    a6
_sqr+0x40:
 $e
_environ:       0xFFFF7DE4
_argc_value:    0x1
float_soft:     0xFFFF0000
_argv_value:    0xFFFF7DEC
_ival:  0x1
_ints:  0x1
__iob:  0x0
__ctype:        0x202020
__bufendtab:    0x0
__smbuf:        0x0
__lastbuf:      0x39D4
_errno: 0x0
__stdbuf:       0x40DC
__sobuf:        0x0
__sibuf:        0x0
_asm_mhfl:      0x0
_end:   0x0
_errnet:        0x0
_edata: 0x1
```

The next request:

```
$r
```

prints out the registers including the program counter and an interpretation of the instruction at that location. The instruction printed for the pc does not always make sense. This is because the pc has been advanced and is either pointing at the next instruction, or is left at a point part way through the instruction that failed. In this case the pc points to the next instruction. In order to find the instruction that failed we could list the instructions and their offsets by the following command.

```
sqr+0x34,5?ia
```

This would show us that the instruction that failed was

```
_sqr+0x3c:move.l d0, (a0)
```

This is the first instruction before the value of the pc. The value printed out for register a0 also indicates that a dereference of its value would fail.

The request:

```
$e
```

prints out the values of all external variables at the time the program crashed.

Setting Breakpoints

Consider the C program in Figure 3. This program, which changes tabs into blanks, is adapted from *Software Tools* by Kernighan and Plauger, pp. 18-27.

Figure 3: C program to decode tabs

```
#include <stdio.h>
#define MAXLINE 80
#define YES             1
#define NO              0
#define TABSP           8

char    input[] = "data";
FILE    *stream;
int     tabs[MAXLINE];
char    ibuf[BUFSIZ];

main()
{
        int col, *ptab;
        char c;

        setbuf(stdout,ibuf);
        ptab = tabs;
        settab(ptab);   /*Set initial tab stops */
        col = 1;
        if((stream = fopen(input,"r")) == NULL) {
                printf("%s : not found\\n",input);
                exit(8);
        }
        while((c = getc(stream)) != EOF) {
                switch(c) {
                        case '\t':              /* TAB */
                                while(tabpos(col) != YES) {
                                        putchar(' ');   /* put BLANK */
                                        col++ ;
                                }
                                break;
                        case '\n':              /*NEWLINE */
                                putchar('\n');
                                col = 1;
                                break;
                        default:
                                putchar(c);
                                col++ ;
                }
        }
```

```
}

/* Tabpos return YES if col is a tab stop */
tabpos(col)
int col;
{
        if(col > MAXLINE)
                return(YES);
        else
                return(tabs[col]);
}

/* Settab - Set initial tab stops */
settab(tabp)
int *tabp;
{
        int i;

        for(i = 0; i<= MAXLINE; i++)
                (i%TABSP) ? (tabs[i] = NO) : (tabs[i] = YES);
}
```

We will run this program under the control of ADB (see Figure 4) by:

```
adb a.out -
```

Breakpoints are set in the program as:

address:b [request]

The requests:

```
settab+4:b
 fopen+4:b
 tabpos+4:b
```

set breakpoints at the starts of these functions. The above addresses are entered as `symbol+4` so that they will appear in any C backtrace since the first instructions of each function is an instruction that links in the new function. Note that one of the functions is from the C library.

Figure 4: ADB output for C program of Figure 3

```
adb a.out -
executable file = a.out
ready
settab+4:b
fopen+4:b
tabpos+4:b
$b
breakpoints
count   bkpt            command
0x1     _tabpos+0x4
0x1     _fopen+0x4
0x1     _settab+0x4
:r
process 19429 created
a.out: running
breakpoint      _settab+0x4:    movem.l #<>,-0x4(a6)
settab+4:d
:c
a.out: running
breakpoint      _fopen+0x4:     jsr     __findiop
$c
_main+0x42:     _fopen  (0x4000, 0x4006)
start+0x48:     _main   (0x1, 0xFFFF7E04)
 tabs/24X
_tabs:          0x1             0x0             0x0             0x0
                0x0             0x0             0x0             0x0
                0x1             0x0             0x0             0x0
                0x0             0x0             0x0             0x0
                0x1             0x0             0x0             0x0
                0x0             0x0             0x0             0x0

 :c
a.out: running
breakpoint      _tabpos+0x4:    movem.l #<>,0x0(a6)
 :s
a.out: running
stopped at      _tabpos+0xA:    moveq   #0x50,d0
 <newline>
a.out: running
stopped at      _tabpos+0xC:   cmp.l   0x8(a6),d0
 <newline>
a.out: running
stopped at      _tabpos+0x10:   bge.s   _tabpos+0x16
 <newline>
a.out: running
stopped at      _tabpos+0x16:   move.l  0x8(a6),d0
 <newline>
```

```
a.out: running
stopped at      _tabpos+0x1A:   asl.l   #0x2,d0
 <newline>
a.out: running
stopped at      _tabpos+0x1C:   add.l   #0x4E50,d0
 <newline>
a.out: running
stopped at      _tabpos+0x22:   move.l  d0,a0
 <newline>
a.out: running
stopped at      _tabpos+0x24:   move.l  (a0),d0
 :d*
 :c
a.out: running
This   is it
process terminated
 settab+4:b settab,5?ia
 tabpos+4,3:b ibuf/20c
 :r
process 19482 created
a.out: running
settab,5?ia
_settab:                link    a6,#0xFFFFFFFC
_settab+0x4:            movem.l #<>,-0x4(a6)
_settab+0xA:            clr.l   -0x4(a6)
_settab+0xE:            moveq   #0x50,d0
_settab+0x12:           cmp.l   -0x4(a6),d0
_settab+0x14:
breakpoint      _settab+0x4:    movem.l #<>,-0x4(a6)
 :c
a.out: running
ibuf/20c
_ibuf:          This
ibuf/20c
_ibuf:          This
ibuf/20c
_ibuf:          This
breakpoint      _tabpos+0x4:    movem.l #<>,0x0(a6)
 $q
process 19482 killed
```

To print the location of breakpoints type:

```
$b
```

The display indicates a *count* field. A breakpoint is bypassed *count-1* times before causing a stop. The *command* field indicates the ADB requests to be executed each time the breakpoint is encountered. In our example no *command* fields are present.

By displaying the original instructions at the function **settab** we see that the breakpoint is set after the instruction to save the registers on the stack. We can display the instructions using the ADB request:

```
settab,5?ia
```

This request displays five instructions starting at **settab** with the addresses of each location displayed.

To run the program simply type:

```
:r
```

To delete a breakpoint, for instance the entry to the function **settab**, type:

```
settab+4:d
```

To continue execution of the program from the breakpoint type:

```
:c
```

Once the program has stopped (in this case at the breakpoint for **fopen**), ADB requests can be used to display the contents of memory. For example:

```
$c
```

to display a stack trace, or:

```
tabs,3/8X
```

to print three lines of 8 locations each from the array called **tabs**. The format **X** is used since integers are four bytes on the MC68000. By this time (at location **fopen**) in the C program, **settab** has been called and should have set a one in every eighth location of **tabs**.

Advanced Breakpoint Usage

When we continue the program with:

```
:c
```

we hit our first breakpoint at **tabpos** since there is a tab following the "This" word of the data. We can execute one instruction by

```
:s
```

and can single step again by hitting "carriage return". Doing this we can quickly single step through **tabpos** and get some confidence that it is working. We can look at twenty characters of the buffer of characters by typing:

```
>buf/20c
```

Several breakpoints of **tabpos** will occur until the program has changed the tab into equivalent blanks. Since we feel that **tabpos** is working, we can remove all the breakpoints by:

```
:d*
```

If the program is continued with:

```
:c
```

it resumes normal execution and continues to completion after ADB prints the message

```
a.out: running
```

It is possible to add a list of commands we wish to execute as part of a breakpoint. By way of example let us reset the breakpoint at **settab** and display the instructions located there when we reach the breakpoint. This is accomplished by:

```
settab+4:b  settab,5?ia
```

It is also possible to execute the ADB requests for each occurrence of the breakpoint but only stop after the third occurrence by typing:

```
tabpos+4,3:b ibuf/20c
```

This request will print twenty character from the buffer of characters at each occurrence of the breakpoint.

If we wished to print the buffer every time we passed the breakpoint without actually stopping there we could type

```
tabpos+4,-1:b ibuf/20c
```

A breakpoint can be overwritten without first deleting the old breakpoint. For example:

```
settab+4:b  settab,5?ia;ptab/o
```

could be entered after typing the above requests. The semicolon is used to separate multiple ADB requests on a single line.

Now the display of breakpoints:

```
$b
```

shows the above request for the **settab** breakpoint. When the breakpoint at **settab** is encountered the ADB requests are executed.

NOTE

Setting a breakpoint causes the value of dot to be changed; executing the program under ADB does not change dot. Therefore:

```
settab+4:b .,5?ia
fopen+4:b
```

will print the last thing dot was set to (in the example **fopen**) not the current location (**settab**) at which the program is executing.

The HP-UX *quit* and *interrupt* signals act on ADB itself rather than on the program being debugged. If such a signal occurs then the program being debugged is stopped and control is returned to ADB. The signal is saved by ADB and is passed on to the test program if:

```
:c
```

is typed. This can be useful when testing interrupt handling routines. The signal is not passed on to the test program if:

```
:c 0
```

is typed.

Other Breakpoint Facilities

Arguments and change of standard input and output are passed to a program as:

```
:r arg1 arg2 ... <infile> outfile
```

This request kills any existing program under test and starts the `a.out` afresh. The process will run until a breakpoint is reached or until the program completes or crashes.

If it is desired to start the program without running it the command

```
:e arg1 arg2 ... <infile> outfile
```

can be executed. This will start the process, and leave it stopped without executing the first instruction.

If the program is stopped at a subroutine call it is possible to step around the subroutine by

```
:S
```

This sets a temporary breakpoint at the next instruction and continues. This may cause unexpected results if `:S` is executed at a branch instruction.

ADB allows a program to be entered at a specific address by typing:

```
address:r
```

The count field can be used to skip the first *n* breakpoints as:

```
,n:r
```

The request:

```
,n:c
```

may also be used for skipping the first *n* breakpoints when continuing a program.

A program can be continued at an address different from the breakpoint by:

```
address:c
```

The program being debugged runs as a separate process and can be killed by:

```
:k
```

All of the breakpoints set so far can be deleted by

```
:d*
```

A subroutine may be called by

```
:x address [parameters]
```

Maps

HP-UX supports several executable file formats. These are used to tell the loader how to load the program file. A shared text program file is the most common and is generated by a C compiler invocation such as `cc pgm.c`. A non-shared text file is produced by a C compiler command of the form `cc -N pgm.c`, while a demand-loaded a.out file is produced by a C compiler command of the form `cc -q pgm.c`. ADB interprets these different file formats and provides access to the different segments through the maps. To print the maps type:

```
$m
```

In nonshared files, both text (instructions) and data are intermixed. In shared files the instructions are separated from data and `?*` accesses the data part of the `a.out` file. The `?*` request tells ADB to use the second part of the map in the `a.out` file. Accessing data in the `core` file shows the data after it was modified by the execution of the program. Notice also that the data segment may have grown during program execution. Figure 5 shows the display of three maps for the same program linked as nonshared, shared, and demand-loaded, respectively. The b, e, and f fields are used by ADB to map addresses into file addresses. The `f1` field is the length of the header at the beginning of the file. The `f2` field is the displacement from the beginning of the file to the data. For a nonshared file with mixed text and data this is the same as the length of the header; for shared files this is the length of the header plus the size of the text portion.

Figure 5: ADB output for maps

```
$ adb a.out.unshared core.unshared
executable file = a.out.unshared
core file = core.unshared
ready
$m
? map   'a.out.unshared'
b1 = 0x0         e1 = 0x19C       f1 = 0x40
b2 = 0x0         e2 = 0x19C       f2 = 0x40
/ map   'core.unshared'
b1 = 0x0         e1 = 0x1000      f1 = 0x3000
b2 = 0xFFFF5000 e2 = 0xFFFF8000 f2 = 0x4000
$v
variables
d = 0x1000
m = 0x107
s = 0x3000
$q
$ adb a.out.shared core.shared
executable file = a.out.shared
core file = core.shared
ready
$m
? map   'a.out.shared'
b1 = 0x0         e1 = 0x18C       f1 = 0x40
b2 = 0x1000      e2 = 0x1010      f2 = 0x1CC
/ map   'core.shared'
b1 = 0x1000      e1 = 0x2000      f1 = 0x3000
b2 = 0xFFFF5000 e2 = 0xFFFF8000 f2 = 0x4000
 $v
variables
b = 0x1000
d = 0x1000
m = 0x108
s = 0x3000
t = 0x1000
 $q
$ adb a.out.demand core.demand
executable file = a.out.demand
core file = core.demand
ready
$m
? map   'a.out.demand'
b1 = 0x0         e1 = 0x18C       f1 = 0x1000
b2 = 0x1000      e2 = 0x1010      f2 = 0x2000
/ map   'core.demand'
b1 = 0x1000      e1 = 0x2000      f1 = 0x3000
b2 = 0xFFFF5000 e2 = 0xFFFF8000 f2 = 0x4000
```

```
 $v
variables
b = 0x1000
d = 0x1000
m = 0x10B
s = 0x3000
t = 0x1000
 $q
```

The `b` and `e` fields are the starting and ending locations for a segment. Given an address, A, the location in the file (either `a.out` or `core`) is calculated as:

$b1 \leq A \leq e1 \rightarrow$ file address = (A-b1)+f1
$b2 \leq A \leq e2 \rightarrow$ file address = (A-b2)+f2

Variables and Registers

ADB provides a set of variables which are available to the user. A variable is composed of a single letter or digit. It can be set by a command such as

```
0x32>5
```

which sets the variable 5 to hex 32. It can be used by a command such as

```
<5=X
```

which will print the value of the variable 5 in hex format.

Some of these variables are set by ADB itself. These variables are:

`0`	last value printed
`b`	base address of data segment
`d`	length of the data segment
`e`	The entry point
`m`	execution type (0x107 (nonshared),0x108 (shared), or 0x10b (demand loaded))
`s`	length of the stack
`t`	length of the text

These variables are useful to know if the file under examination is an executable or `core` image file. ADB reads the header of the core image file to find the values for these variables. If the second file specified does not seem to be a core file, or if it is missing, the header of the executable file is used instead.

Variables can be used for such purposes as counting the number of times a routine is called. Using the example of Figure 3, if we wished to count the number of times the routine `tabpos` is called we could do that by typing the sequence

```
0>5
tabpos+4,-1:b <5+1>5
:r
<5=d
```

The first command sets the variable 5 to zero. The second command sets a breakpoint at `tabpos+4`. Since the count is -1 the process will never stop there but ADB will execute the breakpoint command every time the breakpoint is reached. This command will increment the value of the variable 5 by 1. The `:r` command will cause the process to run to termination, and the final command will print the value of the variable.

`$v` can be used to print the values of all non-zero variables.

The values of individual registers can be set and used in the same way as variables. The command

```
0x32>d0
```

will set the value of the register `d0` to hex 32. The command

```
<d0=X
```

will print the value of the register `d0` in hex format. The command `$r` will print the value of all the registers.

Formatted Dumps

It is possible to combine ADB formatting requests to provide elaborate displays. Below are some examples.

The line:

```
<b,-1/4o4^8Cn
```

prints 4 octal words followed by their ASCII interpretation from the data space of the core image file. Broken down, the various request pieces mean:

`<b`	The base address of the data segment.
`<b,-1`	Print from the base address to the end of file. A negative count is used here and elsewhere to loop indefinitely or until some error condition (like end of file) is detected.

The format `4o4^8Cn` is broken down as follows:

`4o`	Print 4 octal locations.
`4^`	Backup the current address 4 locations (to the original start of the field).
`8C`	Print 8 consecutive characters using an escape convention; each character in the range 0 to 037 is printed as `^` followed by the corresponding character in the range 0140 to 0177. An `^` is printed as `^^`.
`n`	Print a newline.

The request:

```
<b,<d/4o4^8Cn
```

could have been used instead to allow the printing to stop at the end of the data segment (<d provides the data segment size in bytes).

The formatting requests can be combined with ADB's ability to read in a script to produce a core image dump script. ADB is invoked as:

```
adb a.out core < dump
```

to read in a script file, dump, of requests. An example of such a script is:

```
120$w
4095$s
$v
=3n
$m
=3n"C Stack Backtrace"
$C
=3n"C External Variables"
$e
=3n"Registers"
$r
0$s
=3n"Data Segment"
<b,-1/8ona
```

The request 120$w sets the width of the output to 120 characters (normally, the width is 80 characters). ADB attempts to print addresses as:

```
symbol + offset
```

The request 4095$s increases the maximum permissible offset to the nearest symbolic address from 255 (default) to 4095. The request = can be used to print literal strings. Thus, headings are provided in this dump program with requests of the form:

```
=3n"C Stack Backtrace"
```

that spaces three lines and prints the literal string. The request $v prints all non-zero ADB variables. The request 0$s sets the maximum offset for symbol matches to zero thus suppressing the printing of symbolic labels in favor of octal values. Note that this is only done for the printing of the data segment. The request:

```
<b,-1/8ona
```

prints a dump from the base of the data segment to the end of file with an octal address field and eight octal numbers per line.

Figure 7 shows the results of some formatting requests on the C program of Figure 6.

Figure 6: Simple C Program That Illustrates Formatting and Patching

```
char    str1[]  "This is a character string";
int     one     1;
int     number  456;
long    lnum    1234;
float   fpt     1.25;
char    str2[]  "This is the second character string";
main()
{
        one = 2;
}
```

Figure 7: ADB Output Showing Fancy Formats

```
 adb a.out.shared -
executable file = a.out.shared
ready
 <b,-1?8ona
_str1:          052150  064563  020151  071440  060440  061550  060562  060543

_str1+0x10:     072145  071040  071564  071151  067147  0       0       01

_number:
_number:        0       0710    0       02322   037640  0       052150  064563

_str2+0x4:      020151  071440  072150  062440  071545  061557  067144  020143

_str2+0x14:     064141  071141  061564  062562  020163  072162  064556  063400
 <b,20?4o4^8Cn
_str1:          052150  064563  020151  071440  This is
                060440  061550  060562  060543  a charac
                072145  071040  071564  071151  ter stri
                067147  0       0       01      ng@‘@‘@‘@‘@‘@a

_number:        0       0710    0       02322   @‘@‘@aH@‘@‘@dR

_fpt:           037640  0       052150  064563  ? @‘@‘This
                020151  071440  072150  062440   is the
                071545  061557  067144  020143  second c
                064141  071141  061564  062562  haracter
                020163  072162  064556  063400
address not found in a.out file
<b,20?4o4^8t8Cna
```

```
_str1:          052150  064563  020151  071440          This is
_str1+0x8:      060440  061550  060562  060543          a charac
_str1+0x10:     072145  071040  071564  071151          ter stri
_str1+0x18:     067147  0       0       01              ng@‘@‘@‘@‘@‘@a
_number:
_number:        0       0710    0       02322           @‘@‘@aH@‘@‘@dR
_fpt:
_fpt:           037640  0       052150  064563          ? @‘@‘This
_str2+0x4:      020151  071440  072150  062440           is the
_str2+0xC:      071545  061557  067144  020143          second c
_str2+0x14:     064141  071141  061564  062562          haracter
_str2+0x1C:     020163  072162  064556  063400
address not found in a.out file
 <b,a?2b8t^2cn
_str1:          0x54    0x68            Th
                0x69    0x73            is
                0x20    0x69             i
                0x73    0x20            s
                0x61    0x20            a
                0x63    0x68            ch
                0x61    0x72            ar
                0x61    0x63            ac
                0x74    0x65            te
                0x72    0x20            r
$q
```

Patching

Patching files with ADB is accomplished with the **write**, **w** or **W**, request (which is not like the **ed** editor write command). This is often used in conjunction with the **locate**, **l** or **L** request. In general, the request syntax for `l` and `w` are similar as follows:

```
?l value
```

The request **l** is used to match on two bytes, `L` is used for four bytes. The request `w` is used to write two bytes, whereas `W` writes four bytes. The **value** field in either `locate` or `write` requests is an expression. Therefore, decimal and octal numbers, or character strings are supported.

In order to modify a file, ADB must be called as:

```
adb -w file1 file2
```

When called with this option, `file1` is created if necessary and opened for both reading and writing. `file2` can be opened for reading but not for writing.

For example, consider the C program shown in Figure 6. We can change the word "This" to "The " in the executable file for this program, **ex7**, by using the following requests:

```
adb -w ex7 -
?l 'Th'
?W 'The '
```

The request `?l` starts at dot and stops at the first match of "Th" having set dot to the address of the location found. Note the use of `?` to write to the `a.out` file. The form `?*` would have been used for a shared text file.

More frequently the request will be typed as:

```
?l 'Th'; ?s
```

and locates the first occurrence of "Th" and print the entire string. Execution of this ADB request will set dot to the address of the "Th" characters.

As another example of the utility of the patching facility, consider a C program that has an internal logic flag. The flag could be set by the user through ADB and the program run. For example:

```
adb a.out -
:e arg1 arg2
flag/w 1
:c
```

The `:e` request is used to start `a.out` as a subprocess with arguments `arg1` and `arg2`. If there is a subprocess running ADB writes to it rather than to the file so the `w` request causes `flag` to be changed in the memory of the subprocess.

Anomalies

Below is a list of some strange things that users should be aware of.

1. Function calls and arguments are put on the stack by the `link` instruction. Putting breakpoints at the entry point to routines means that the function appears not to have been called when the breakpoint occurs.
2. If a `:s` command is executed at a branch instruction, and the branch is taken, the command will act as a `:c` command. This is because a breakpoint is set at the next instruction and if is not reached, the process will not stop.

Command Summary

Formatted Printing

? *format*	print from *a.out* file according to *format*
/ *format*	print from *core* file according to *format*
= *format*	print the value of **dot**
?w *expression*	write expression into *a.out* file
/w *expression*	write expression into *core* file
?l *expression*	locate expression in *a.out* file

Breakpoint and Program Control

:b	set breakpoint at **dot**
:c	continue running program
:d	delete breakpoint
:k	kill the program being debugged
:r	run `a.out` file under ADB control
:s	single step

Miscellaneous Printing

$b	print current breakpoints
$c	C stack trace
$e	external variables
$f	floating registers
$m	print ADB segment maps
$q	exit from ADB
$r	general registers
$s	set offset for symbol match
$v	print ADB variables
$w	set output line width

Calling the Shell

!	call *shell* to read rest of line

Assignment to Variables

>*name*	assign dot to variable or register *name*

Format Summary

a	the value of dot
b	one byte in hexadecimal
c	one byte as a character
d	two bytes in decimal
f	four bytes in floating point
i	MC68000 instruction
o	two bytes in octa
n	print a newline
r	print a blank space
s	a null terminated character string
*n***t**	move to next *n* space tab
u	two bytes as unsigned integer
x	hexadecimal
Y	date
^	backup dot
"..."	print string

Expression Summary

Expression Components

decimal integer	e.g. 0d256
octal integer	e.g. 0277
hexadecimal	e.g. 0xff
symbols	e.g. flag _main
variables	e.g. <b
registers	e.g. <pc <d0
(expression)	expression grouping

Dyadic Operators

+	add
–	subtract
*	multiply
%	integer division
&	bitwise *and*
\|	bitwise *or*
#	round up to the next multiple

Monadic Operators

~	not
*	contents of location
–	integer negate

Notes

Index

a

adb command 1
adb command format 2
anomolies 26

b

breakpoints:
 facilities 16–17
 setting 9–13
 using 14–15

c

command summary 27

d

data formats, display 3–5
debugging:
 C programs 6–8
 core image 6–8
display array contents 3
display data formats 3–5
display requests, information 3
dumps, formatted 21

e

executable file formats 17

f

file patching 25
format:
 adb command 2
 data display 3–5
 executable file 17
formatted dumps 21

i

information display requests 3
invoking adb 1

m

maps used to access executable file segments 17

p

patching files 25

r

registers and variables, use of 20

s

scrolling through memory 3
setting breakpoints 9–13
summary:
 command 27
 expression 29
 format 28

v

variables and registers, use of 20

Table of Contents

C Debugger (cdb)

Part 1: Introduction

Tutorial Text Conventions 1
Overview of cdb 2
Overview of Interprocess Debugging 3
Compiling Programs 4
Debugger Command Conventions 6
Notational Conventions 6
Variable Name Conventions 7
Expression Conventions 9
Procedure Call Conventions 11
Running cdb 12
Example Program 14

Part 2: Viewing Commands

File Code Viewing Commands 15
Print Current File, Procedure and Line Number 15
Change Files and Print First Executable Line 16
Print Groups of Lines 16
Print Window of Text 17
Move Forward/Backward from Current Line 18
Miscellaneous File Viewing Commands 19
Stack Viewing Commands 20
Trace Stack for Expr Levels 20
Set Viewing Location 21
Data Viewing Commands 22
Print Variable's Value 22
View Non-current Location Variables 23
List Command 23
Miscellaneous Data Viewing Commands 24
Display Formats 25

Part 3: Job Control Commands

Run/Terminate the Program 27
Terminate Current Child Process 28
Continue After Breakpoint/Signal 29
Single Step After Breakpoint 30

Part 4: Breakpoint Commands
Set a Breakpoint 34
List Breakpoints 37
Delete Breakpoints 37
Miscellaneous Breakpoint Commands 38

Part 5: Assertion Control Commands and Signal Handling Commands
Assertion Control Commands 41
Create New Assertion 41
Modify an Assertion 42
Tracing Program Execution 43
Toggle the State 44
Delete All Assertions 44
Signal Handling Commands 45
Reverse Handling of Signal 46

Part 6: Record, Playback, and Other cdb Commands
Record and Playback Commands 49
Miscellaneous Record and Playback Commands 52
Other Commands 53

C Debugger (cdb)
Part 1: Introduction

Tutorial Text Conventions

The following conventions are used throughout this tutorial:

- Italics indicate files and HP-UX commands, system calls, and subroutines found in the *HP-UX Reference* manual as well as titles of manuals. Italics are also used for symbolic items either typed by you or displayed by the system as discussed below. Examples include */usr/lib/nls/american/prog.cat, date(1),* and *pty(4).* The parenthetic number shown for commands, system calls, and other items found in the *HP-UX Reference* is a convention used in that manual.
- **Boldface** is used when a word is first defined and for **general emphasis**.
- Computer font indicates a literal typed by you or displayed by the system. A typical example is:

  ```
  cdb main.c
  ```

 Note that when a command or file name is part of a literal, it is shown in computer font and not italics. However, if the command or file name is symbolic (but not literal), it is shown in italics as the following example illustrates:

 `cdb` *executable_file*

 In this case you would type in your own *executable_file*. If the command has optional arguments, they are designated by square brackets, **[]**, as the example below shows:

 [*line*]**p**[*count*]

- Unless otherwise stated, all references such as "see the *ptrace(2)* entry for more details" refer to entries in the *HP-UX Reference* manual. If you cannot find an entry where you expect it to be, use the *HP-UX Reference* manual's *Permuted Index*.

Overview of cdb

cdb is a symbolic source-level debugger that provides a controlled execution environment for C, FORTRAN, and Pascal programs. This tool can be used to debug C, FORTRAN, or Pascal programs without needing to know internals.

The scenario in which you use *cdb* is: if you have problems in your program, you re-compile the program and then use *cdb* to assist in finding and correcting errors.

This tutorial describes the commands needed to use *cdb*. The tutorial provides a description of the commands and each command's syntax. There are programming examples in which the more important commands are used.

There are certain hardware dependencies, symbol table dependencies, diagnostics, warnings, and bugs associated with *cdb*. The authoritative reference on these items is the manual page *cdb(1)* in the *HP-UX Reference*. Everything else about *cdb* is detailed or implied in its pages, along with a quick reference of all the commands presented in this tutorial.

Overview of Interprocess Debugging

Both *cdb* and *adb* are **interprocess** debuggers. Interprocess debuggers run separately from the programs processes being debugged.

In HP-UX, *cdb* (and *adb*) interact with the program being debugged through *ptrace(2)*. This intrinsic allows a parent process read and write memory, and register locations in a child process, as well as causes the child process to machine-instruction step, continue (**free run**), and terminate.

The debugger *cdb* is the **parent process** and the program being debugged is the **child process**. In this document the terms **child process**, **target program**, **your program**, and **program being debugged** are synonymous.

Compiling Programs

The C, FORTRAN, and Pascal compilers emit debugging information when you compile with the *-g* option. This debug information is massaged by the assembler (Pascal and FORTRAN bypass the assembler) and the linked output ends up in the executable program file. *cdb* needs this information to be able to debug your program. If you want to use *cdb* on a particular procedure, it **must** be compiled with the *-g* option. You don't have to compile your entire program with *-g* (it's usually easier to do it that way), but as a minimum, the **main** procedure must be compiled with *-g* (otherwise *cdb* can't debug your program).

```
cc -g program
```

Compiling with *-g* increases the size of your executable file considerably (for example, compiling *cdb* source with *-g* leads to a 6x increase). However, the memory requirements will not change appreciably because the debug data is not loaded into memory.

The *objectfile* is the executable program file which has had one or more of its component modules compiled with debug option(s) (for example, *-g*) turned on. The *-g* option causes the linker to append */usr/lib/end.o* to your *objectfile*. This support module */usr/lib/end.o* must be included as the last object file in the list of those linked, except for libraries included with the *-l* option of *ld(1)*. The */usr/lib/end.o* subroutine contains buffer space used by *cdb* during command line procedure calls. An increase of 200 bytes in memory requirements is caused by compiling with *-g*. (Some systems automate this; see the *cdb(1)* "Hardware Dependencies" section.) The default for *objectfile* is *a.out*.

The *corefile* is a core image from a failed execution of *objectfile*. The default for *corefile* is *core*. (Note: the Series 500 does not support corefiles.)

The options available are:

`-d` *dir* — names an alternate directory where source files are located. They are searched in the order given. If a source file is not found in any alternate directory, the current directory is searched last.

`-r` *file* — names a *record* file which is invoked immediately (for overwrite, not for append). See the section below entitled "Record and Playback Commands" for a description of this feature.

`-p` *file* — names a *playback* file which is invoked immediately. See the section below entitled "Record and Playback Commands" for a description of this feature.

`-S` *num* sets the size of the string cache to *num* bytes. The default *num* depends on the symbol table format used. The option is not available for all formats. The string cache holds data read from *objectfile*.

There can only be one *objectfile* and one *corefile* per debugging session (activation of the debugger). The program (*objectfile*) is not invoked as a child process until you give an appropriate command (see the "Job Control Commands" chapter). The same program may be restarted, as different child processes, many times during one debugging session.

This debugger is a complex, interactive tool with many synergistic and combinatorial features. What you can do with it is often limited only by your imagination. Remember, however, that the debugger is only a **window** into the world consisting mostly of the program being debugged and the system it runs on. If something puzzling happens, you may need to consult a manual which describes the program or the system, in order to understand the behavior.

Debugger Command Conventions

The debugger remembers the current file, current procedure, current line, and current data location. They are a function of what you have been viewing (not necessarily executing) most recently. Many commands use these current locations as defaults; many commands set them as a side effect. It is important to keep this in mind when deciding what a command does in any particular situation.

For example, if you stop in procedure *asub*, then view procedure *bsub*, then ask for the value of local variable *i*, the debugger assumes that the variable belongs to procedure *bsub*.

Notational Conventions

Most commands are of the form [*modifier*] *command-letter* [*options*]. Numeric modifiers before and after commands can be any numeric expression. They need not be just simple numbers. A blank is required before any numeric *option*. Multiple commands on one line must be separated by ;.

These are common modifiers and other special notations:

`(A \| B \| C)`	Any one of A or B or C is required.
`[A \| B \| C]`	Any one of A or B or C is optional.
file	A file name.
proc	A procedure (or function, or subroutine) name.
var	A variable name.
number	A specific, constant number (e.g. 9, not 4+5). Floating point (real) numbers may be used any place a constant is allowed.
expr	Any expression, but with limitations stated below.
depth	A stack depth, as printed by the *t* command. The top procedure is at a *depth* of zero. A negative *depth* acts like a *depth* of zero. Stack depth usually means **exactly at the specified depth**, not the **first instance at or above the specified depth**.
format	A style for printing data. See the "Viewing Commands" chapter for details.

commands A series of debugger commands, separated by ;, entered on the command line, or saved with a breakpoint or assertion. Semicolons are ignored (as commands) so they can be freely used as command separators. Commands may be grouped with {} for the *a*, *b*, *if*, and *!* commands. In all other cases, commands inside {} are ignored.

Variable Name Conventions

Variables are referenced exactly as they are named in your source file(s). Case sensitivity is controlled by the *Z* command. Be careful with one letter variable names, since they can be confused with commands. If an expression begins with a variable that might be mistaken for a command, just enclose the expression in () (e.g. *(k)*), or eliminate any white space between the variable and the first operator (use *k= 9* instead of *k = 9*).

If you are interested in the value of some variable *var*, there are a number of ways of getting it, depending on where and what it is:

var Search the stack for the most recent instance of the current procedure. If found, see if *var* is a parameter or local variable of that procedure. If not, search outward using scoping rules for *var*.

proc.var Search the stack for the most recent instance of *proc*. If found, see if it has a parameter or local variable named *var*, as before.

proc.depth.var Use the instance of *proc* that is at depth *depth* (exactly), instead of the most recent instance. This is very useful for debugging recursive procedures where there are multiple instances on the stack.

:var Search for a global (not local) variable named *var*.

Dot is shorthand for the last thing you viewed (see the "Data Viewing Commands" section). It has the same size it did when you last viewed it. For example, if you look at a **long** as a **char**, then . is considered to be one byte long. This is useful for treating things in unconventional ways, such as changing the second highest byte of a **long** without changing the rest of the **long**. *Dot* may be treated like any other variable.

NOTE

The . (*dot*) is the **name** of this magic location. If you use it, it is dereferenced like any other name. If you want the **address** of something that is, say, 30 bytes farther on in memory, do not use .+30. That would take the contents of *dot* and add 30 to it. Instead, say &.+30, which adds 30 to the **address** of *dot*.

Special variables are names for things that are not normally directly accessible. Special variables include:

`$`*var*	The debugger has room in its own address space for a number of user-created special variables. They are all of type **long**, and do not take on the type of any expression they are assigned to. Names are defined when they are first seen. For example, saying `$xyz = 3*4` creates special symbol *$xyz*, and assigns to it the value 12. Special variables may be used just like any other variables.
`$pc`, `$fp`, `$sp`, `$r0`, etc.	These are the names of the program counter, the frame pointer, the stack pointer, the registers, etc. To find out which names are available on your system, use the *l r* (list registers) command. All registers act as type **integer**.
`$result`	This is used to reference the return value from the last procedure exit. Where possible, it takes on the type of the procedure. *$short* and *$long* are available as alternate ways of looking at *$result*.
`$signal`	This lets you see and modify the current child process signal number.
`$lang`	This lets you see and modify the current language (0 for C, 1 for FORTRAN, or 2 for Pascal).
`$line`	This lets you see and modify the current source line number, which can be set with a number of different commands.
`$malloc`	This lets you see the current amount of memory (bytes) allocated at run-time for use by the debugger itself.

`$cBad` This lets you see and modify the number of machine instructions the debugger will step while in a non-debuggable procedure before setting an up-level breakpoint and free-running to it. Setting it to a small value can improve debugger performance, at the risk of taking off free-running after missing the up-level break for some reason.

To see all the special variables, including the predefined ones, use the *l s* (list specials) command.

You can also look up code addresses with:

proc#line

which searches for the given procedure name and line number (which must be an executable line within *proc*) and uses the code address of that line. Just referring to a procedure *proc* by name uses the code address of the entry point to that procedure.

Expression Conventions

Every expression has a value, even simple assignment statements, as in C. *Naked* expression values (those which aren't command modifiers) are always printed unless the next token is **;** (command separator) or **}** (command block terminator). Thus breakpoint and assertion commands (see the appropriate sections below) are normally silent. To force an expression result to be printed, follow the expression with */n* (print in normal format; see below).

Integer constants may begin with *0* for octal or *0x* or *0X* for hexadecimal. They are **int** if they fit in two bytes, **long** otherwise. If followed immediately by *l* or *L*, they are forced to be of type **long** (this is useful on systems where **int** is two bytes).

Floating point constants must be of the form:

digits.*digits*[`e | E | d | D | L | l` [+|-]*digits*]

for example, `1.0`, `3.14e8`, or `26.62D-31`. One or more leading digits is required to avoid confusion with `.` (*dot*). A decimal point and one or more following digits is required to avoid confusion for some command formats. If the exponent doesn't exactly fit the pattern shown, it is not taken as part of the number, but as separate token(s). The *d* and *D* exponent forms are allowed for compatibility with FORTRAN. The *l* and *L* exponent forms are allowed for compatibility with Pascal. However, all floating point constants are taken as doubles, regardless.

Character constants must be entered in single quotes (for example, 'n') and are treated as **integers**. C string constants must be entered in double quotes (for example, "Hello World") and are treated like *char ** (i.e., pointer to **char**). FORTRAN and Pascal strings may be enclosed in either single quotes '' or double quotes "". Character and string constants may contain the standard backslashed escapes understood by the C compiler and the *echo(1)* command, including \b, \f, \n, \r, \t, \\, \', and \nnn. However, \ RETURN is not supported, in quotes or at the end of a command line.

Expressions are composed of any combination of variables, constants, and C operators. If the debugger is invoked as *cdb*, the C operator *sizeof* is also available. If the debugger is invoked as *fdb*, FORTRAN operators are also available and FORTRAN meanings take precedence where there is a conflict. The same is true for Pascal if the debugger is invoked as *pdb*.

If there is no active child process and no *corefile*, you can only evaluate expressions containing constants.

Expressions approximately follow the C rules of promotion, e.g. **char**, **short**, and **int** become **long**, and **float** becomes **double**. If either operand is a **double**, floating point math is used. If either operand is **unsigned**, unsigned math is used. Otherwise, normal (integer) math is used. Results are then cast to proper destination types for assignments.

If a floating point number is used with an operator that doesn't normally permit it, the number is cast to **long** and used that way. For example, the C binary operator ~ (bit invert) applied to the constant *3.14159* is the same as ~*3*.

Note that = means **assign** except in Pascal. In Pascal, = is a comparison operator; use := for assignments. For FORTRAN use == or **.EQ.**. For example, if you invoke the debugger as *cdb*, then set `$lang = 2` (Pascal), you must say `$lang := 0` to return to C.

Use // for division, instead of /, to distinguish from display formatting (see the " Data Viewing Commands" section).

The special unary operator *$in* (not to be confused with debugger local variables) evaluates to 1 (true) if the operand is an address inside a debuggable procedure and *$pc* (the current child process program location) is also in that procedure, else it is 0 (false). For example, *$in main* is true if the child process is stopped in *main()*.

If the first expression on a line begins with + or −, use () around it to distinguish from the + and − commands (see the "Data Viewing Commands" section). Parentheses may also be needed to distinguish an expression from a command it modifies.

You can attempt to dereference any constant, variable, or expression result using the C * operator. If the address is invalid, an error is given.

Whenever an array variable is referenced without giving all its subscripts, the result is the address of the lowest element referenced. For example, consider an array declared as `x[5][6][7]` in C, `x(5,6,7)` in FORTRAN, or `x[1..5,2..6,3..7]` in Pascal. Referencing it simply as `x` is the same as just `x` in C, the address of `x(1,1,1)` in FORTRAN, or the address of `x[1,2,3]` in Pascal. Referencing it as `x[4]` is the same as `&(x[4][0][0])` in C, the address of `x(1,1,4)` in FORTRAN, or the address of `x[4,2,3]` in Pascal.

If a not-fully-qualified array reference appears on the left side of an assignment, the value of the right-hand expression is stored into the element at the address specified.

String constants are stored in a buffer in the file */usr/lib/end.o*. The debugger starts storing strings at the beginning of this buffer, and moves along as more assignments are made. If the debugger reaches the end of the buffer, it goes back and reuses it from the beginning. In general this doesn't cause any problems. However, if you use very long strings, or if you assign a string constant to a global pointer, problems could arise.

Procedure Call Conventions

Procedures may be invoked from the command line, even within expressions. For example:

```
xyz = $abc * (3 + def (ghi - 1, jkl, "Hi Mom"))
```

calls procedure *def* when its value is needed in the expression.

Any breakpoints encountered during command line procedure invocation are handled as usual. However, the debugger has only one active command line at a time. If it stops in a called procedure for any reason, the remainder (if any) of the old command line is tossed, with notice given.

If you attempt to call a procedure when there is no active child process, one is started for you as if you gave a single-step command first. Unfortunately, this means that the data in *corefile* (if any) may disappear or be reinitialized.

If you send signal SIGINT (e.g., hit the [BREAK] key) while in a called procedure, the debugger aborts the procedure call and returns to the previous stopping point (the start of the main program for a new process).

You can call any procedure that is in your *objectfile*, even if it is not debuggable (was not compiled with the *-g* option). For example, assume that you reference *printf()* in your program, so the code for it is in your *objectfile*. Then you can enter on the command line:

```
printf ("This works! %d %c\n", 9, '?');
```

If you wonder what procedures are available, do a list labels command (*l l*). If you want to have some library routines available for debugging, but they aren't referenced anywhere in your code (so they aren't linked), relink your program with the *-u* option to force their inclusion.

Note that procedure name *_end_* is declared in *end.c*.

Running cdb

If an *a.out* file exists, then you invoke *cdb* by typing:

```
cdb
```

Otherwise, you need to specify an executable file as shown below.

To invoke the debugger on your C program, type:

cdb *executable_file*

Run the debugger on FORTRAN programs via:

fdb *executable_file*

and on Pascal programs via:

pdb *executable_file*

/bin/fdb and */bin/pdb* are links to */bin/cdb*. The *cdb* debugger does some language-dependent processing based on how it was invoked (*cdb*, *fdb*, or *pdb*). Examples of this are:

- FORTRAN arrays (column-major storage)
- FORTRAN *CHAR** and Pascal *string* variables
- Pascal *PACKED arrays of CHAR*

You may change the **current** language from within *cdb/fdb/pdb* with the *$lang* special variable (see the "Conventions" section for more details).

Throughout the remainder of this document, *cdb* will be used as a generic term for *cdb/fdb/pdb*.

The *cdb* debugger needs to be able to access the source files for your program. The debugger assumes they are in the current directory. If they're not, use the *-d* command line option to specify their location. For example:

```
cdb -d src1 -d src2 bin/pgm
```

runs *cdb* on *./bin/pgm* with source in *./src1* and *./src2*.

The *cdb* debugger starts up by displaying file and procedure counts and then the first executable line of your program. At this point your program has **not** been loaded into memory.

cdb then prompts for commands with the > character.

Example Program

The example program used throughout this tutorial is listed below. Almost all the *cdb* commands covered in this tutorial will be illustrated using these two files (*main.c* and *sub.c*). Type them in exactly as shown, using an appropriate text editor (e.g., *vi*). Then compile them both and start *cdb*.

For the purposes of this document, the file *main.c* must contain the main program:

```
main ()
{
    long i;
    i = 5;
    asub(i);
}
```

The file *sub.c* must contain the subroutines:

```
asub (arg)
long arg;
{
    bsub(arg);
}

bsub (myarg)
long myarg;
{
    /* do nothing */
}
```

To compile these program files, use the C compiler and the *-g* option as shown in the previous section "Compiling Programs". Type:

```
cc -g main.c sub.c
```

During compilation, two object files *main.o* and *sub.o* will be created and placed in the current directory. You can use the *lsf* command to check for them. To start the debugger on the **default executable object file** *a.out* type:

```
cdb  a.out
```

NOTE

All examples in this tutorial were run on a Series 500 computer. Addresses will differ from those on Series 200 or 300 machines.

Part 2: Viewing Commands

A user can view the source code statically (before the program has executed) or dynamically (during execution). The stack and data, on the other hand, are meaningless until the program is executing and a breakpoint is reached.

File Code Viewing Commands

One must understand the concept of **current** lines, files, and procedures in order to use *cdb*. The *cdb* debugger interprets everything relative to the **current** viewing location; this holds particularly to line numbers and variable names.

Print Current File, Procedure and Line Number

Syntax:

```
e
```

Example:

```
$cdb a.out
Source files:     3
Procedures:       4
main.c: main: 4: i=5;
>e
main.c: main: 4: i=5;
```

This command prints the line you are presently located at within the file. It shows the current file, procedure, line number, and source line (*main.c: main: 4: i = 5;*). Commands that show the file and procedure with a source line, skip (do not print) any leading white space from the source line.

Change Files and Print First Executable Line

Syntax:

e *file*
e *proc*

This command places you in the file or procedure designated. Entering a file sets the current line number to 1. Entering a procedure sets the current file and line to the first executable line of the procedure. You can enter **any** file and look at it from *cdb*; it does not have to be a program source file.

Example:

```
>e sub.c
sub.c: 1: asub (arg)
>e asub
sub.c: asub: 4: bsub (arg);
```

Notice that the second *e* command places you into the *sub.c* file at the first executable line of *asub()*. To return to *main.c* simply type:

```
>e main.c
main.c: 1: main()
```

Print Groups of Lines

Syntax:

[*line*]p [*count*]

The *p* command can be used several ways. When *p* is used alone, the current line is output. Using *p* with just *line* prints the line specified by that number. If a *count* follows the *p*, *count* lines will be printed starting at *line*. *p* followed only by *count*, prints from the current line forward *count* lines. If more than one line is printed, the current line is marked with a = in the leftmost position.

Example:

```
>p
    1: main ()
>5p
    5:      asub(i);
>p2
    5:      asub(i);
=   6:  }
>2p 3
    2:  {
    3:      long i:
=   4:      i = 5;
```

Print Window of Text

Syntax:

[*line*] `w` [*window size*]
[*line*] `W` [*window size*]

Instead of using *p* to print sections of text, sometimes the *w* and *W* commands are more useful. The window commands are used for quickly scrolling through source files (or any file). These commands print blocks of text thereby reducing the need to refer to paper listings during a debugging session. Window commands (*w* defaults to 11 lines and *W* defualts to 21 lines) print the block of text centered around the current line (or any specified line). The *line* parameter specifies the current line number. Then *window size* designates how many lines around the current line are printed.

You can cause the previous *w* or *W* command to be repeated by pressing RETURN. This causes the next successive block of text to be displayed. The *cdb* debugger remembers the size and direction of text windowing for the next RETURN command.

Example:

```
>e sub.c
sub.c: 1: asub (arg)
>5 p
    5:  }
>w
    1:  asub (arg)
    2:  long arg;
    3:  {
    4:      bsub(arg);
=   5:  }
    6:
    7:   bsub (myarg)
    8:   long myarg;
    9:   {
   10:         /* do nothing */
   11:   }
>4 w
    1:  asub (arg)
    2:  long arg;
    3:  {
=   4:      bsub(arg);
    5:  }
    6:
    7:   bsub (myarg)
    8:   long myarg;
    9:   {
   10:         /* do nothing */
   11:   }
```

```
>w 5
    2:  long arg;
    3:  {
=   4:      bsub(arg);
    5:  }
    6:
>9 w 5
    7:   bsub (myarg)
    8:   long myarg;
=   9:   {
   10:         /* do nothing */
   11:   }
```

Move Forward/Backward from Current Line

Syntax:

+[*lines*]
-[*lines*]

This command moves the cursor *lines* forward when you use + and *lines* backward when you use the -. The default is 1.

Example:

```
>- 3
     6:
>+ 4
    10:       /*do nothing*/
```

The window command and these directional commands can be blended to build the *+/- W/w* commands which are useful for changing direction. The *-W* and *-w* commands cause the preceding block of text to be displayed. While *+W* and *+w* cause the following block of text to be displayed.

Miscellaneous File Viewing Commands

`dir` *directory*	Add *directory* to the list of alternate source directories. The effect is the same as using the *-d* invocation option. If the file containing the main procedure does not reside in the current directory, its directory must be specified with the *-d* option.
L	This is a synonym for *0E* (see the "Set Viewing Location" section).
line	Print source line number *line* in the current file.
+w[*size*] *+W*[*size*]	Print a window of text, of the given or default *size*, beginning at the end of the previous window, if the previous command was a window command, or at the current line otherwise.
-w[*size*] *-W*[*size*]	Print a window of text, of the given or default *size*, ending at the beginning of the previous window, if the previous command was a window command, otherwise end at the current line.

If after any window command you give a *w* or *W* command with no *line* specified, the debugger prints the following window of source text; or if the previous window command was *-w* or *-W* the previous window is printed, using the given *size* (or the default if none). Pressing RETURN after any window command does the same thing, but uses the previous *size* as well.

/[*string*] Search forward through the current file, from the line after the current line, for *string*.

?[*string*] Search backward for *string*, from the line before the current line.

Searches wrap around the end or beginning of the file, respectively. If *string* is not specified, the previous one is used. **Wild cards** and regular expressions are not supported; *string* must be literal. **Case sensitivity** is contolled by *z*; the default is **insensitive** (see the section "Other Commands" for details).

n	Repeat the previous / or **?** command using the same *string* as previously.
N	The same as *n*, but the search goes in the opposite direction as specified by the previous / or **?** command.

These search commands, /, *?*, *n*, and *N* work the same as in *vi(1)*.

Stack Viewing Commands

These commands are only meaningful after the child process stops (e.g., on a breakpoint) because there is nothing on the stack until the child process is running. The procedure calling chain is displayed with the *t* and *T* commands.

A detailed description for using and setting breakpoints is provided in the "Breakpoint Commands" section. For this example type:

```
>b                                          (set the breakpoint)
Added:
  1: count:  1  asub: 4: bsub(arg);
>r                                          (run the program)
Starting process 1246: "a.out"

breakpoint at 0x60180006
sub.c: asub: 4: bsub(arg);
```

Trace Stack for Expr Levels

Syntax:

[*depth*] `t`
[*depth*] `T`

The *t* command traces the stack for the first *depth* (default 20) level and displays the procedures on the stack and their parameter values. The *T* supplements this information with local variables which are also displayed, using the */n* format (except that arrays and pointers are shown as addresses, and only the first word of structures is shown).

Example:

```
>t
0  asub (arg = 5)      [sub.c: 4]
1  main ()    [main.c: 5]
2  start  +0x0000001a (0x1, 0xc0000030, 0xc0000040)
3  unknown ()
>T
0  asub (arg = 5)      [sub.c: 4]
1  main ()    [main.c: 5]
        i          = 5
2  start  +0x0000001a (0x1, 0xc0000030, 0xc0000040)
3  unknown ()
```

Non-debuggable procedures are also displayed but their parameters are displayed in hexadecimal.

Set Viewing Location

Syntax:

[*depth*]E

The *E* command sets the current viewing location to the procedure on the stack at depth *depth* and prints the current file name, procedure name, and line. The point of suspended execution is at *depth* = 0. For example, with the above stack trace the command `1E` sets the current viewing line to line 5 in *main.c* which is the call to *asub()*.

The *E* command only sets the **viewing** location. This means that using *E* to set the location to a prior instance of a recursive procedure and then querying the value of variable *x* will show *x* in the most recent instance of the procedure. The *proc.depth.var* syntax must be used in this case.

The *E* command is handy for quickly looking at the source code for the calling chain (perhaps to determine the context of the current procedure call). You use *0E* or its synonym *L* to get back to the point of suspended execution after roaming around setting breakpoints or viewing other files, etc.

Example:

```
>E
sub.c: asub: 4: bsub(arg);
>1E
main.c: main: 5 +0x0000000c: asub(i);
>0E
sub.c: asub: 4: bsub(arg);
```

Data Viewing Commands

Print Variable's Value

Syntax:

expr
expr/format
expr?format

The *expr* can be as simple as the name of a variable in a child process; or it can be a complex combination of variables and arithmetic operators. See the "Expression Conventions" section for further discussion. The debugger returns the value of the variable designated by *expr*. It is handled as if you had typed **expr/n** (print expression in normal format), unless followed by ; or }, in which case nothing is printed.

All the variables in *expr* must be known in the current viewing location. For example, if you try to query the value of *arg* when the current location is not in *asub()*, you will recieve the error message *Unknown name or command "arg"*.

If there is a conflict between a variable name and a command, the command name takes precedence. To query the value of such a variable, either enclose the name in parentheses, or specify a format. For example, *i* in *main()* conflicts with the *if* command:

Example:

```
>arg
arg = 5
>e main.c
main.c: 1: main()
>i
Missing "{"
>(i)
i = 5
```

Sometimes during debugging it is necessary to print the contents of a variable using a different *format* than the normal default format (*n*). In the example below *i* is printed out in decimal as an integer. There are a variety of *formats* available (see "Miscellaneous Data Viewing Commands" and "Display Formats"). The / specifies printing the value of the *expr* and the ? designates printing the address of the *expr*. Then ^ indicates backing up to the preceding location while the . reverses the direction again to forward.

```
>i/d
i = 5
>i?d
-1073741424
>^/d
0xc000018c 56
>./d
0xc000018c 56
```

View Non-current Location Variables

Syntax:

proc.var
proc.depth.expr

With these forms you can view variables in a procedure not containing the current viewing location or look at a variable at a particular depth on the procedure stack (useful for recursive programs).

Example:

```
>asub.arg
arg = 5
>asub.1.arg
Procedure "asub" not found at stack depth 1
>main.1.i
i = 5
```

List Command

Syntax:

l[*proc*[.*depth*]]
l (a | b | d | z)
l (f | g | l | p | r | s) [*string*]

This command *l* lists all parameters and local variables of the current procedure or the specified *proc* (if given) at the specified *depth* (if any). Data is displayed using */n* format, except that all arrays and pointers are shown simply as addresses and only the first word of any structure is shown.

The letters in parentheses stand for **a**ssertions, **b**reakpoints, **d**irectories (where to search for files), **z**ignals (signal actions), **f**iles (sourcefiles), **g**lobal variables (known to linker), **l**abels, **p**rocedure names, **r**egisters, or **s**pecial variables. If *string* is present, only those things with the same initial characters are listed.

Example:

```
>l main
i          = 5
>l a
No assertions
>l b
 1: count: 1  asub: 4: bsub(arg);
>l f
 0:  main.c        0x60100000 to 0x60100019
 1:  sub.c         0x60180000 to 0x60180023
 2:  end.c         0x60200000 to 0x60200007
>l p as
 1: asub           0x60180000 to 0x60180015
```

Miscellaneous Data Viewing Commands

expr/format

Print the contents (value) of *expr* using *format*. For example, `abc/x` prints the contents of *abc* as an **integer**, in hexadecimal.

expr?format

Print the address of *expr* using *format*. For example, `abc?o` prints the address of *abc* in octal.

^[[/]*format*]

Back up to the preceding memory location (based on the size of the last thing displayed). Use *format* if supplied, or the previous *format* if not. Note that no / is needed after the ^. Also note that you can reverse direction again (e.g., start going forward) by entering . (*dot*), which is always an alias for the current location, followed by RETURN.

Display Formats

Display formats are used only with Data Viewing Commands. The *format* is of the form: [*][*count*]*formchar*[*size*].

* means **use alternate address map** (e.g., *abc*), if maps are supported.

The *count* is the number of times to apply the format style *formchar*. It must be a **number** not an expression.

The *size* is the number of bytes to be formatted for each *count*, and overrides the default *size* for the format style. It must be a positive decimal *number* (except short hand notations, see below). The *size* is disallowed with those *formchar*'s where it makes no sense.

For example, `abc/4x2` prints, starting at the memory location of *abc*, four two-byte numbers in hexadecimal.

Using an optional upper-case letter with formats that print numbers has the same affect as appending the *l* option to the format (see below). For example, *O* prints 4 bytes in octal (i.e. **long**). These formats, which are useful on systems where **integer** is shorter than **long**, are noted below. The following formats are available:

n	Print in the *normal* format, based on the type. Arrays of **char** and pointers to **char** are interpreted as strings, and structures are fully dumped.
(*d D*)	Print in decimal (as **integer** or **long**).
(*u U*)	Print in unsigned decimal (as **integer** or **long**).
(*o O*)	Print in octal (as **integer** or **long**).
(*x X*)	Print in hexadecimal (as **integer** or **long**).
(*b B*)	Print a byte in decimal (either way).
(*c C*)	Print a character (either way).
(*e E*)	Print in *e* floating point notation (as **float** or **double**) (see *printf* (3)). Remember that floating point constants are always doubles.
(*f F*)	Print in *f* floating point notation (as **float** or **double**).
(*g G*)	Print in *g* floating point notation (as **float** or **double**).
a	Print a string using *expr* as the address of the first byte.

s	Print a string using *expr* as the address of a pointer to the first byte. This is the same as saying **expr/a*, except for arrays.
t	Show the type of *expr* (usually a variable or procedure name). For true procedure types you must actually call the procedure, (e.g., `def(2)/t`; alone *def* is the address of the function, i.e., an integer).
p	Print the name of the procedure containing address *expr*.
S	Do a formatted dump of a structure (only with symbol tables which support it). Note that *expr* must be the address of a structure, not the address of a pointer to a structure.

There are some shorthand notations for *size*:

b	1 byte (**char**).
s	2 bytes (**short**).
l	4 bytes (**long**).

These can be appended to *formchar* instead of a numeric *size*. For example, `abc/xb` prints one byte in hexadecimal.

If you view an object with a *size* (explicitly or implicitly) less than or equal to the size of a **long**, the debugger changes the basetype to something appropriate for that *size*. This is so `.` (*dot*) works correctly for assignments. For example, `abc/c2` sets the type of `.` to **short**. One side effect is that if you look at a **double** using a **float** format, *dot* loses accuracy or has the wrong value.

Part 3: Job Control Commands

The parent (*cdb* debugger) and the child (*objectfile*) processes take turns running. The debugger is only active while the child process is stopped due to a signal, including hitting a breakpoint, or terminated for whatever reason.

Run/Terminate the Program

Syntax

```
R
r[arguments]
```

Use *R* to run a new child process with no *argument* list and *r* to run a new child process with a given *argument* list (or the previous list if none is given). The existing child process, if any, is terminated first.

The *r* command is the most versatile way to begin program execution. The *arguments* list can contain < and > for redirecting standard input and standard output. (< does an *open(2)* of file descriptor 0 for read-only; > does a *creat(2)* of file descriptor 1 with mode 0666). The *arguments* list may contain shell variables and metacharacters, quote marks, or other special syntax. Special shell syntax is expanded by a Bourne shell. Because {} are shell metacharacters, *r* cannot be safely saved in a breakpoint or assertion command list.

If no *arguments* are given, the ones used with the last *r* command are used again. No arguments are used if *R* was used last. For example, the command line:

```
>r arg1 arg2 arg3 >file1 <file2
```

passes *arg1*, *arg2*, and *arg3* as arguments and redirects *stdin* and *stdout*. It is equivalent to running your program from the shell as in:

```
program arg1 arg2 arg3 >file1 <file2
```

The *r* command expands shell variables and meta-characters before passing the argument string to the child process. Remember, it always kills off an existing child process first. You can do this manually with the *k* command, too (see example under the "Terminate" section).

Where the *r* command starts your program and lets it free run, the *R* command works similarly, except no arguments or I/O redirection can be specified.

Example:

```
>r
Starting process 942: "a.out"

breakpoint at 0x60180006
sub.c: asub: 4: bsub(arg);
>r arg1 arg2
Terminating process 942
Starting process 947: "a.out arg1 arg2"

breakpoint at 0x60180006
sub.c: asub: 4: bsub(arg);
>R
Terminating process 947
Starting process 948: "a.out"

breakpoint at 0x60180006
sub.c: asub: 4: bsub(arg);
```

Whenever *cdb* stops and displays a line of your program, that line has not been executed yet. So setting a breakpoint (see the "Breakpoint Commands" section) on a line will cause *cdb* to stop before executing any code for the statement(s) on that line.

Terminate Current Child Process

Syntax:

```
k
```

Terminate (**kill**) the current child process if one exists.

Example:

```
>k
Really terminate child process? y
Terminating process 948
```

Continue After Breakpoint/Signal

Syntax:

[*count*]c[*line*]
[*count*]C[*line*]

The *c* command causes execution to continue after a breakpoint or signal, while ignoring the signal, if any. The *C* command allows the signal, if any, to be received. This is fatal to the child process if it does not catch or ignores the signal.

There are two fields associated with a breakpoint: *count* and *command*. The *count* field is discussed here; the *command* field is explained later in the "Breakpoint Commands" section. The *count* field associated with a breakpoint is the number of times the breakpoint is encountered prior to recognition. If the *count* is positive, the breakpoint is **permanent** and *count* decrements with each encounter. When *count* goes to zero, the breakpoint is recognized and the *count* is reset to one. If *count* is negative, the breakpoint is **temporary** and *count* increments with each encounter. Once *count* is zero, the breakpoint is recognized, then deleted.

NOTE

Count is set to -1 (temporary) or 1 (permanent) for any new breakpoint. Only then can it be modified by the continue (*c*) command.

The *line*, if given, designates a temporary breakpoint at that line number, with a count of -1.

Example:

```
>r
Starting process 942: "a.out"

breakpoint at 060180006
sub.c: asub: 4: bsub(arg);
>c 11          **temporary breakpoint**
Added:
 2: count: -1 (temporary)  bsub: 11:  }

breakpoint at 0x60180022
sub.c: bsub: 11:  }
>c
Child process terminated normally
>e main
main.c: main: 4:  i = 5;
>5
    5:      asub(i);
>b
```

```
Added:
 2: count: 1   main: 5: asub(i);
>r
Starting process 1029: "a.out"

breakpoint at 0x601000a
main.c: main: 5: asub(i);
>C

breakpoint at 0x60180006
sub.c:  asub: 4: bsub(arg);
```

Single Step After Breakpoint

Syntax:

[*count*]**s**
[*count*]**S**

If there is no child process currently active, you can **step** into your program with the *s* and *S* commands. These commands start your program and then stop before the first executable line of the main procedure.

With these two commands, you can execute your program a source line at a time. The *s* command traces debuggable procedure calls and enters the debuggable procedure. It single steps 1 (or *count*) statements. Successive RETURN's repeat with a *count* of 1. If *count* is less than one, the child process is not stepped. Note that the child process continues with the current signal, if any. (You can set *$signal = 0* to prevent this.)

If you accidently step down into a procedure you don't care about, use the *bU* command to set a temporary up-level breakpoint, and then continue using *c*.

The *S* command steps over procedure calls because *cdb* detects the occurrence of a procedure call and plants a temporary breakpoint at the point of return, free runs the program until that breakpoint is hit, then machine-instruction steps to the next source line boundary. If a breakpoint is hit during execution of the called procedure, execution stops at that point and the temporary breakpoint is deleted.

Stepping into a non-debuggable procedure (i.e., one that hasn't been compiled with *-g*) with *s* will cause behavior equivalent to *S*. In general, you can't do anything with non-debuggable code. In the stepping case, *cdb* recognizes that it has stepped into an unknown (non-debuggable) procedure, so it sets an invisible up-level breakpoint and free runs the child.

You can't specify *arguments* with *s* and *S*. If you need to specify *arguments* to redirect I/O, the easiest way is to set a breakpoint on the first line of *main()* and execute with *r*.

Example:

```
>D
All breakpoints deleted.
>s
Starting process 1089:  "a.out"
main.c: main: 4: i = 5;
>s
main.c: main: 5: asub(i);
>s
sub.c:  asub: 4: bsub(arg);
>S
sub.c: asub: 5: }
>2s
main.c: main: 6: }
Child process terminated normally
```

The debugger has no knowledge about or control over child processes forked in turn by the process being debugged. Also, it gets very confused (leading to **bad access messages**) if the process being debugged executes a different program via *exec(2)*.

Child process output may be (and usually is) buffered. Hence it may not appear immediately after you step through an output statement such as *printf(3)*. It may not appear at all if you kill the process.

Notes

Part 4: Breakpoint Commands

The debugger provides a number of commands for setting and deleting breakpoints. A breakpoint has three attributes associated with it:

- *address* - All the commands which set a breakpoint are simply alternate ways to specify the breakpoint address. The breakpoint is then encountered whenever the instruction *address* is about to be executed, regardless of the path taken to get there. Only one breakpoint at a time (of any type or count) may be set at a given *address*. Setting a new breakpoint at *address* replaces the old one, if any.

- *count* - The number of times the breakpoint is encountered prior to recognition. If *count* is positive, the breakpoint is **permanent**, and *count* decrements with each encounter. Each time *count* goes to zero, the breakpoint is recognized, and *count* is reset to one (so it stays there until explicitly set to a different value by a *c* or *C* command).

 If *count* is negative, the breakpoint is **temporary**, and *count* increments with each encounter. Once *count* goes to zero, the breakpoint is recognized, then deleted.

 A *count* of zero is used internally by the debugger and means that the breakpoint is deleted when the child process next stops for any reason, whether it hit that breakpoint or not. Commands saved with such breakpoints are ignored. Normally you never see this kind of breakpoints.

 Note that *count* is set to either -1 (temporary) or 1 (permanent) for any new breakpoint. It can then be modified only by the *c* or *C* command.

- *commands* - *cdb* commands which are executed when a breakpoint is recognized. These are separated by ; and may be enclosed in {} to delimit the list saved with the breakpoint from other commands on the same line. If the first character is anything other than {, or if the matching } is missing, the rest of the line is saved with the breakpoint.

 Remember that the results of expressions followed by ; or } are not printed unless you specify a print format. You can use */n* (normal format) to force printing of a result.

 Saved commands are not parsed until the breakpoint is recognized. If *commands* does not exist then, after recognition of the breakpoint, the debugger waits for command input.

 The debugger has only one active command line at a time. When it begins to execute breakpoint commands, the remainder (if any) of the old command line is tossed, with notice given.

Breakpoints can be set at executable statements only. By definition an **executable line** is one for which the compiler has emitted an **SLT** (Source Line Table) entry. The C compiler emits SLT entries for each logical statement (*assignment, while, for, if*, etc). If you put several assignment statement on the same source line, the compiler will emit several SLT entries for that line. You can set breakpoints only at the first SLT entry for a line, but stepping through that line with *s* will repeatedly show the same line. This is because you are hitting addresses corresponding to successive SLT entries on that line.

Attempting to set a breakpoint on a non-executable line has several possible results. If the line is before the first executable line in a procedure or after the last executable line in a file, *cdb* displays:

```
"Can't set breakpoint (invalid address)"
```

If the line is between two executable lines, *cdb* **rounds forward** and sets the breakpoint on the following executable line.

Set a Breakpoint

Syntax:

[*line*] `b` [*commands*]

cdb provides several commands for setting breakpoints. The simplest is *b* which sets a permanent breakpoint at the current line. The *commands* descriptor is a list of *cdb* commands, separated by semi-colons, which are executed when the breakpoint is hit. The *line* number refers to the current file. If the *line* number is omitted, the breakpoint is set on the current line.

When the breakpoint is recognized, *commands* are executed. If there are none, the debugger pauses for command input. If immediate continuation is desired, finish the command list with *c*.

For example, suppose you want to set a breakpoint in some file or procedure other than where you are at the moment. First, use the *e* command to get you to the right file or procedure. Look around for the line where you want the break to occur (using searches, or just by printing the lines). Once you are there, you can just say *b* to set a breakpoint on that line.

So to set a breakpoint in *asub()*, you must first set the current file to *sub.c*. Do this with the *e* command previously discussed:

```
e sub.c
```

or

```
e asub
```

Then set the breakpoint with the *b* command, possibly specifying a line number.

Example:

```
>e asub
sub.c: asub: 4: bsub(arg);
>b
Added:
 1: count: 1  asub: 4: bsub(arg);
>
```

or:

```
>e sub.c
sub.c: 1: asub (arg)
>4b
Added:
 1: count: 1  asub: 4: bsub(arg);
>D                              (to delete the breakpoint)
```

You can specify commands to be executed when a breakpoint is hit. Consider the following example in which *b t;c* plants a breakpoint in *bsub()* to print a stack trace, then continue execution:

```
>e bsub
sub.c: bsub: 11: }
>b t;c
Added:
 1: count: 1  bsub: 11: }
    {t;c}
>r
Starting process 3981:  "a.out"

breakpoint at 0x60180022
sub.c: bsub: 11: }
 0 bsub (myarg = 5)    [sub.c: 11]
 1 asub (arg = 5)    [sub.c: 4]
 2 main ()    [main.c: 5]
 3 start +0x0000001a (0x1, 0xc0000030, 0xc0000040)
 4 unknown ()
Child process terminated normally
```

You can suppress the printing of the location by using the *Q* command (**quiet**) as the first in the list. .If the *quiet* command appears as the first command in a breakpoint's command list, the normal announcement of *proc: line: text* is not made. This allows quiet checks of variables, etc. to be made without cluttering up the screen with unwanted output. The *Q* command is ignored if it appears anywhere else. Here's the same example as above, except it uses the *Q* command:

```
>e bsub
sub.c: bsub: 11: }
>b Q;t;c
Added:
 1: count: 1  bsub: 11: }
    {Q;t;c}
>r
Starting process 22980:  "a.out"
 0 bsub (myarg = 5)    [sub.c: 11]
 1 asub (arg = 5)    [sub.c: 4]
 2 main ()    [main.c: 5]
 3 start +0x0000001a (0x1, 0xc0000030, 0xc0000040)
 4 unknown ()
Child process terminated normally
```

There are several more breakpoint setting commands with a variety of uses; they are listed below in "Miscellaneous Breakpoint Commands".

List Breakpoints

Syntax:

```
B
l b
```

Both forms list all breakpoints in the format `num: count: nnn proc: ln: contents`, followed by {*commands*} (see the example). The leftmost number is an index number for use with the *d* (delete) command.

Example:

```
>B
 1: count: 1 bsub:  11: }
    (Q;t;c)
>l b
 1: count: 1 bsub:  11: }
    (Q;t;c)
```

Delete Breakpoints

Syntax:

```
D[b]
[expr] d
D p
```

D deletes all breakpoints including **procedure** breakpoints. You can delete breakpoints one-by-one with the *d* command.

The version *d* deletes the breakpoint at the current line or the breakpoint number *expr*. If *expr* is absent, delete the breakpoint at the current line, if any. If there is none, the debugger executes a *B* command instead. Be careful; the breakpoints may be renumbered after each *d* command.

The *D p* command deletes all **procedure** breakpoints. All breakpoints set by commands other than *bp* will remain set.

Example:

```
>D
All breakpoints deleted
>4
    4:      bsub(arg);
>b
Added:
 1: count: 1 asub: 4 bsub(arg);
>d
Deleted:
 1: count: 1 asub: 4 bsub(arg);
>b
Added:
 1: count: 1 asub: 4 bsub(arg);
>11
    11:  }
>b
Added:
 2: count: 1 bsub: 11:  }
>2d
Deleted:
 2: count: 1 bsub: 11:  }
>Dp
No procedure breakpoints
```

Miscellaneous Breakpoint Commands

bp[*commands*]

Set permanent breakpoints at the beginning (first executable line) of every debuggable procedure. When any procedure breakpoint is hit, *commands* are executed.

It is permissible to set other permanent or temporary breakpoints at the same locations as these **procedure** breakpoints. If a procedure and non-procedure breakpoint are both hit at the same location, the non-procedure breakpoint has priority; the effect is the same as if there were no procedure breakpoint. It is not possible to alter the *count* of a procedure breakpoint. Procedure breakpoints must be activated and deleted as a group; it is not possible to set or delete individual ones.

Procedure breakpoints are useful for procedure stepping and tracing. For example, the command:

```
bp Q;1t;c
```

sets up procedure tracing by printing the current procedure at each breakpoint.

For the following commands, if the second character is upper case, e.g. *bU* instead of *bu*, then the breakpoint is temporary (*count* is -1), not permanent (*count* is 1).

[*depth*]`bb`[*commands*]
[*depth*]`bB`[*commands*]

Set a breakpoint at the beginning (first executable line) of the procedure at the given stack *depth*. If *depth* is not specified, use the currently viewed procedure, which might not be the same as the one at *depth* zero.

[*depth*]`bx`[*commands*]
[*depth*]`bX`[*commands*]

Set a breakpoint at the exit (last executable line) of the procedure at the given stack *depth*. If *depth* is not specified, use the currently viewed procedure, which might not be the same as the one at *depth* zero. The breakpoint is set at a point such that all returns of any kind go through it.

[*depth*]`bu` [*commands*]
[*depth*]`bU` [*commands*]

Set an up-level breakpoint. The breakpoint is set immediately after the return to the procedure at the specified stack *depth* (default one, not zero). A *depth* of zero means current location, e.g. `0bU` is a way to set a temporary breakpoint at the current value of `$pc`.

[*depth*]`bt`[*proc*][*commands*]
[*depth*]`bT`[*proc*][*commands*]

Trace current procedure (or procedure at *depth*, or *proc*). This command sets breakpoints at both the entrance and exit of a procedure. By default, the entry breakpoint *commands* are `Q;2t;c`, which show the top two procedures on the stack and continues. The exit breakpoint is always set to execute `Q;L;c`, which prints the procedure's return value and continues.

If *depth* is given, *proc* must be absent or it is taken as part of *commands*. If *depth* is missing but *proc* is specified, the named procedure is traced. If both *depth* and *proc* are omitted, the current procedure is traced, which might not be the same as the one at *depth* zero.

If *commands* are present, they are used for the entrance breakpoint, instead of the default shown above.

address `ba`[*commands*]
address `bA`[*commands*]

Set a breakpoint at the given code address. Note that *address* can be the name of a procedure or an expression containing such a name. Of course, if the child process is stopped in a non-debuggable procedure, or in prologue code (before the first executable line of a procedure), things may seem a little strange.

The next two commands, while not strictly part of the breakpoint group, are used almost exclusively as arguments to breakpoints (or assertions).

if [*expr*]
{*commands*}[{*commands*}]

If *expr* evaluates to a non-zero value, the first group of commands (the first {} block) is executed, else it (and the following {, if any) is skipped. In general, all other {} blocks are always ignored (skipped), except when given as an argument to an *a*, *b*, or **!** command. The *if* command is nestable, and may be abbreviated to *i*.

"any string you like"

Print the given string, which may have the standard backslashed character escapes in it, including \n for newline. This command is useful for labelling output from breakpoint commands.

Part 5: Assertion Control Commands and Signal Handling Commands

Assertion Control Commands

Assertions are lists of commands that are executed **before every statement**. This means that, if there is even one active assertion, the program is single stepped at the machine-instruction level. In other words, it runs very slowly. The primary use for assertions is tracking down nasty bugs, that result from someone corrupting a global variable. Each assertion is individually activated or suspended, in addition to the overall assertions mode.

Create New Assertion

Syntax:

a [*commands*]

To create a new assertion with a given *commands* list, which is not parsed until it's executed, use the *a* command. As with breakpoints, the *commands* list may be enclosed in {} to delimit it from other commands on the same line. Use the *l a* command to list all current assertions and the overall mode.

The debugger has only one active command line at a time. When it begins to execute assertion commands, the remainder (if any) of the old command line is tossed, with notice given.

Example:

```
>a if ($in main) {L;i/n}
Overall assertions state: ACTIVE
  0: Active  {if ($in main) {L; i/n}}
```

This code sets an assertion that checks if the next executable statement is in *main()*. If that statement is in *main()*, then it is displayed, along with the value of *i* in normal format. If the next executable statement is not in *main()*, nothing is displayed.

Modify an Assertion

Syntax:

[*expr*] `a (a | d | s)`

Modify the assertion numbered *expr*: **a**ctivate it, **d**elete it, or **s**uspend it. Suspended assertions continue to exist, but have no effect until reactivated.

Example:

```
>e main
main.c:  main: 4: i = 5;
>a if ($in main) {L;i/n|}
Overall assertions state: ACTIVE
   0: Active  {if ($in main) {L; i/n|}}
>0ad                                    (delete the mistyped assertion above)
Assertion 0 deleted
>l a
No assertions
>a if ($in main) {L;i/n}
Overall assertions state: ACTIVE
   0: Active  {if ($in main) {L; i/n}}
>r
Starting process 27700:  "a.out"
main.c: main: 4: i = 5;
i = 0
main.c: main: 5: asub(i);
i = 5
main.c: main: 6: }
i = 5
Child process terminated normally
>0ad
Assertion 0 deleted
```

The *a* command can be used to trace variable values. For example, it can be used to trace the variable *i* which is in *main* but not known in *asub()* or *bsub()*.

Example:

```
a if (abc != $abc) {$abc = abc; abc/d; if (abc > 9) {x}}
```

This command sets up an assertion to report the changing value of some global variable (*abc*), and to stop if it ever exceeds some value. It uses a debugger local variable (*$abc*) to keep track of the value of *abc*.

Tracing Program Execution

a L

Syntax:

```
a  L
```

This just traces execution a line at a time until something happens (e.g., you hit the BREAK key). Output from running program with above assertion. The example below illustrates setting a flag indicating whether *bsub()* has been called. It echos the flag value at every statement.

Example:

```
>a L;if ($in bsub) {$bsubcalled=1}; $bsubcalled/n
Overall assertions state:  ACTIVE
 0:  Active     {L;if ($in bsub) {$bsubcalled=1}; $bsubcalled/n}
>$bsubcalled=0
$bsubcalled = 0
>r
Starting process 27718:  "a.out"
main.c: main: 4: i = 5;
$bsubcalled = 0
main.c: main: 5: asub(i);
$bsubcalled = 0
sub.c: asub: 4: bsub(arg);
$bsubcalled = 0
sub.c: bsub: 11: }
$bsubcalled = 1
sub.c: asub: 5: }
$bsubcalled = 1
main.c: main: 6: }
$bsubcalled = 1
Child process terminated normally
```

Toggle the State

Syntax:

```
A
```

Toggle the overall state of the assertions mechanism between *active* and *suspended*.

Example:

```
>A
Assertions are SUSPENDED
>r
Terminating process 1299
Starting process 1300: "a.out"

breakpoint at 0x6010000a
main.c: main: 5: asub(i);
>A
Assertions are ACTIVE
>r
Terminating process 1299
Starting process 1300: "a.out"
main.c: main: 4: i = 5;
```

Delete All Assertions

Syntax:

```
D a
```

Delete all assertions.

Example:

```
>D a
All assertions deleted
```

Certain commands (*r*, *R*, *c*, *C*, *s*, *S*, and *k*) are not allowed while assertions are running. They must appear after the *x*, if at all (see "Display Formats").

Signal Handling Commands

The debugger catches all signals bound for the child process before the child process sees them. (This is a function of the *ptrace(2)* mechanism.) For many signals, this is reasonable. Most processes are not set up to handle segmentation errors, etc. Other processes do quite a bit with signals and the constant need to continue from a signal catch can be tedious. It is possible to alter this behavior for any or all signals.
There are three signal action attributes in the debugger:

- *cdb* can have the child process ignore or not ignore a signal. This determines whether the child process sees the signal.
- *cdb* can report or not report on when a child process receives a signal. For example, *cdb* prints out the line it occurred on.
- *cdb* can stop or not stop when a child process receives a signal.

Each above attribute is independent of the other two, yet six combinations are legal.

For this section a different program is required since *main.c* does not send or receive signals. Type this new program in the file *sig.c*:

```
main()
{
    long i,j;
    i = 5;
    j = i/0;
 }
```

To compile and run the program type:

```
cc -g -o sig sig.c
sig
```

Start the *cdb* debugger by entering:

```
cdb sig
```

Reverse Handling of Signal

Syntax

[*signal*] `z [i][r][s][Q]`

The *z* command maintains the *zignal* (signal) handling table. The variable *signal* is a valid signal number (the default is the current signal). The options (which must be all one word) toggle the state of the appropriate flag: ignore, report, or stop. If *Q* is present, the new state of the signal is not printed.

The sequence *l z* is used to list the current handling of all signals. The sequence `8 z` will only report on signal 8. Note that just *z* with no options tells you the state of the current or selected signal.

To toggle the state of a signal, type *signal* `z` and the actions to toggle. For example, assuming a start up state of: do stop, don't ignore, and do report, the command `8 z sir` tells the debugger to not stop, do ignore and do not report on signal 8. The command `8 z ir` toggles *Ignore* to *No* and *Report* to *Yes*. Doing `8 z ir` again toggles the flags back to the previous state.

When the debugger ignores a signal, the child process does not receive that signal.

Example:

```
cdb sig
Source files:  2
Procedures:    2
sig.c: main: 4: i = 5;
>l z
Sig   Stop   Ignore   Report   Name
  1   Yes    No       Yes      hangup
  2   Yes    Yes      Yes      interrupt
  ...
  8   Yes    No       Yes      floating point exception
  ...
  19  Yes    No       Yes     power fail
>8 z                                    (list current state of signal 8)
Sig   Stop   Ignore   Report   Name
  8   Yes    No       Yes      floating point exception
>r
Starting process 11827: "sig"

floating point exception (no ignore) at 0x6010000e
sig.c: main: 5 +0x00000004: j = i/0;
>8 z sir                                (reverse the handling of signal 8)
Sig   Stop   Ignore   Report   Name
  8   No     Yes      No       floating point exception
>r
```

```
Terminating process 11827
Starting process 11873: "sig"
Child process terminated normally
>8 z ir
Sig   Stop   Ignore   Report   Name
  8   No     No       Yes      floating point exception
>r
Starting process 11891: "sig"

floating point exception (no stop) (no ignore) at 0x6010000e
sig.c: main: 5 +0x00000004: j = i/0;
floating point exception (core dumped) (no ignore) at 00000000
(file unknown): unknown: (line unknown)
Child process terminated on signal
>8 z ir
Sig   Stop   Ignore   Report   Name
  8   No     Yes      No       floating point exception
```

Notes

Part 6: Record, Playback, and Other cdb Commands

Record and Playback Commands

The debugger supports a record-and-playback feature to help re-create program states and to record all debugger output. It is particularly useful for bugs requiring long setups. With playback, you can automatically re-create a program state that may take a long time to reconstruct.

The -r (record) and -p (playback) options specify record and playback files that the debugger will use. The example below sets up a scenario similar to that in the "Tracing Program Execution" section, with several other command lines entered after the command `cdb a.out -r record1`.

Example:

```
cdb a.out -r record1
Source files:   3
Procedures:     4
Recording is ON, overwriting "record1"
main.c: main: 4: i=5;
>a L;if ($in bsub) {$bsubcalled=1}; $bsubcalled/n
Overall assertions state:  ACTIVE
 0:  Active     {L;if ($in bsub) {$bsubcalled=1}; $bsubcalled/n}
>$bsubcalled=0
$bsubcalled = 0
>r
Starting process 27718:  "a.out"
main.c: main: 4: i = 5;
$bsubcalled = 0
main.c: main: 5: asub(i);
$bsubcalled = 0
sub.c: asub: 4: bsub(arg);
$bsubcalled = 0
sub.c: bsub: 11: }
$bsubcalled = 1
sub.c: asub: 5: }
$bsubcalled = 1
main.c: main: 6: }
$bsubcalled = 1
Child process terminated normally
>e asub
sub.c: asub: 4: bsub(arg);
>b t;c
Added:
 2: count: 1 asub: 4: bsub(arg);
    {t;c}
>l b
```

```
 1: count:  0 (temporary) start +0x00000024: (line unknown)
 2: count: 1 asub: 4: bsub(arg);
    {t;c}
>A
Assertions are SUSPENDED
>l a
Overall assertions state: SUSPENDED
 0: Active      {L;if ($in bsub)  {$bsubcalled=1}; $bsubcalled/n}
>q
Really quit? y
$
```

All these commands are now saved in the file *record1*:

```
a L;if ($in bsub) {$bsubcalled=1}; $bsubcalled/n
$bsubcalled=0
r
e asub
b t;c
l b
A
l a
q
y
```

Cdb can then be exited and returned to by:

```
cdb a.out -p record1
```

The debugger re-runs all the commands and thereby re-creates the original environment.

You can also save the instructions from inside *cdb* using the > *record1* command as the first statement of the session. The sequence of commands typed in immediately after is saved in *record1*. Now instead of quiting *cdb*, the record file can be closed and started as a playback using >c followed by < *record1*. The commands saved in *record1* are then re-executed and the results printed to the screen. The << command plays back *record1* in single step mode and provides the specialized set of instructions (see the example).

Example:

```
cdb sig
Source files: 2
Procedures:   2
sig.c main: 4: i = 5;
> > record1
Recording is ON, overwriting "record1"
>b t;c
Added:
  1: count: 1 main: 4: i = 5;
     {t,c}
>r
Starting Process 13597: "a.out"

breakpoint at 0x60100006
main.c: main: 4: i = 5;
 0 main ()     [main.c: 4]
 1 start +0x0000001a (0x11, 0xc0000030, 0xc0000040)
 2 unknown ()
Child process terminated normally
> > c
Closing record file "record1"
>  < record1                    (start playback from file "record1")
Playing back from "record1"
b t;c
Added:
  1: count: 1 main: 4: i = 5;
     {t,c}
r
Starting Process 13597: "a.out"

breakpoint at 0x60100006
main.c: main: 4: i = 5;
 0 main ()     [main.c: 4]
 1 start +0x0000001a (0x11, 0xc0000030, 0xc0000040)
 2 unknown ()
Child process terminated normally
>c
End of playback
> << record1                   (single step, with instructions, playback)
Playing back from "record1"
b t;c (<cr>, S, <num>, C, Q, or ?): ?
<cr>  execute one command line;
S     skip one command line;
<num> execute number of command lines;
C     continue through all playback;
Q     quit playback mode.
1
Deleted:
```

```
 1: count: 1 main: 4: i = 5;
    {t;c}
Added:
 1: count: 1  main: 4: i = 5;
    {t;c}
r  (<cr>, S, <num>, C, Q, or ?): Q
End of playback
```

Miscellaneous Record and Playback Commands

The rest of the record and playback commands are used in the same manner with slight variations. The syntax and a brief decription of each is listed below:

>*file*	This command sets or changes the recordfile to *file* and turns recording on. Any previous contents of *file* are overwritten. Only commands are recorded to this file.
>>*file*	The same as >*file*, but appends to *file* instead of overwriting.
>@*file*	
>>@*file*	Set or change record-all file to *file*, for overwriting or appending. The record-all file may be opened or closed independently of (in parallel with) the recordfile. All debugger standard output is copied to the record-all file, including prompts, commands entered, and command output. However, child process output is not captured.
>(t \| f \| c)	Turn recording on (t) or off (f), or close the recording file (c). When recording is resumed, it appends after commands recorded earlier. In this context, >> is the same as >.
>@(t \| f \| c)	Turn record-all on, off, or close the record-all file. In this context, >>@ is the same as >@.
>	Display the current recording status. >> does the same thing.
>@	Display the current record-all status. >>@ does the same thing.

Only command lines read from the keyboard or a playback file are recorded in the **recordfile**. For example, if recording is turned on in an assertion, it doesn't take affect until assertion execution stops. Both the commands and resulting output are recorded in the **record-all file**.

Command lines beginning with >, <, or ! are not copied to the current recordfile (but they are copied to the record-all file). You can override this by beginning the lines with blanks.

Other Commands

Two options that were not covered previously are:

- *-S* — size of cache option sets the size of the string cache to the given number of bytes, instead of the default.
- *-u* — unique names option tells the debugger to expect names in the symbol table to start with an extra underscore.

Each of the following commands are fairly straightforward. Therefore, only the syntax and a brief description of each is provided:

RETURN	Repeat the previous command
^D	To repeat one command 10 times use CTRL D
! [*command_line*]	This shell escape invokes a shell program in the same manner as *vi(1)*.
f [*printf-style-format*]	Set address printing format (the default is reset), using *printf(3)* format specifications (**not** debugger format styles).
h \| **help**	Print the debugger help file (command summary).
I	Print information (inquire) about the state of the debugger.
q	Quit the debugger. To be sure you don't lose a valuable environment, this command requests confirmation.
Z	Toggle case sensitivity in searches. This affects everything: file names, procedure names, variables, and string searches! The debugger starts out as **not** case sensitive.
g *line*	Go to an address in the procedure on the stack at *depth* zero (not necessarily the same as the currently viewed procedure).

Notes

Index

a

activate assertions 44
assertion control commands 41ff
assertions, activate, suspend, or delete 44

b

bp command 38
breakpoint commands 33ff
breakpoint/signal, continue after 29

c

cache, string, size of 53
change files and print first executable line 16
command conventions 6
commands, record/playback 49ff
compiling programs with debugging option 4
continue after breakpoint/signal 29
conventions:
 command 6
 expression 9
 notational 6
 procedure call 9
 variable name 7
corefile 4
create new assertion 41
current line, move forward or backward from 18

d

Data Viewing Commands 22ff
debugger options 4
delete all assertions 44
delete breakpoints 37
dir command 19
display formats 25ff

e

E command 21
e command 16
example program 14
expression conventions 9

f

file code viewing commands 15

g

-g compiler option required to debug programs 4

h

handling of signal, reverse 46

i

invoking cdb 12

j

job control commands 27ff

k

kill current child process 28

l

L command 19
l command 23
list breakpoints 37
list command 23
location variables, view non-current 23

m

memory requirements 4
modify and assertion 42
move forward or backward from current line 18

n

notational conventions 6

o

objectfile 4
options, debugging 4

p

p command 16
parent/child relationship of cdb and program 3
playback/record commands 49ff
print current file, procedure, and line number 15
print first executable line, change files and 16
print groups of lines 16
print variable value 22
print window of text 17
procedure call conventions 9
ptrace 45
ptrace(2) 3

r

record/playback commands 49ff
reverse signal state 46
run program 27

s

searches 19
set breakpoint 34
set permanent breakpoints at start of each procedure 38
set viewing location 21
signal handling commands 45ff
signal state, reverse 46
signal/breakpoint, continue after 29
single-step after breakpoint 30
size of string cache 53
stack viewing commands 20ff
stop program 27
suppress printing of breakpoint location 36
suspend assertions 44

t

T command 20
t command 20
terminate current child process 28
terminate program 27
text, print window of 17
toggle signal state 46
toggle state of assertions mechanism 44
trace program execution 43
trace stack 20

u

unique names option 53

v

variable name conventions 7
variable value, print 22
view non-current location variables 23
viewing commands 15ff

w

w and W commands 17
window of text, print 17

Table of Contents

Make a Program for Maintaining Computer Programs

Introduction 2
Basic Features 3
Description Files and Substitutions 6
Command Usage 8
Implicit Rules 10
Example 12
Suggestions and Warnings 14
Appendix. Suffixes and Transformation Rules 15

Make a Program for Maintaining Computer Programs

In a programming project, it is easy to lose track of which files need to be reprocessed or recompiled after a change is made in some part of the source. *Make* provides a simple mechanism for maintaining up-to-date versions of programs that result from many operations on a number of files. It is possible to tell *Make* the sequence of commands that create certain files, and the list of files that require other files to be current before the operations can be done. Whenever a change is made in any part of the program, the *Make* command will create the proper files simply, correctly, and with a minimum amount of effort.

The basic operation of *Make* is to find the name of a needed target in the description, ensure that all of the files on which it depends exist and are up to date, and then create the target if it has not been modified since its generators were. The description file really defines the graph of dependencies; *Make* does a depth-first search of this graph to determine what work is really necessary.

Make also provides a simple macro substitution facility and the ability to encapsulate commands in a single file for convenient administration.

Introduction

It is common practice to divide large programs into smaller, more manageable pieces. The pieces may require quite different treatments: some may need to be run through a macro processor, some may need to be processed by a sophisticated program generator (such as *Yacc* or *Lex*). The outputs of these generators may then have to be compiled with special options and with certain definitions and declarations. The code resulting from these transformations may then need to be loaded together with certain libraries under the control of special options. Related maintenance activities involve running complicated test scripts and installing validated modules. Unfortunately, it is very easy for a programmer to forget which files depend on which others, which files have been modified recently, and the exact sequence of operations needed to make or exercise a new version of the program. After a long editing session, one may easily lose track of which files have been changed and which object modules are still valid, since a change to a declaration can obsolete a dozen other files. Forgetting to compile a routine that has been changed or that uses changed declarations will result in a program that will not work, and a bug that can be very hard to track down. On the other hand, recompiling everything in sight just to be safe is very wasteful.

The program described in this report mechanizes many of the activities of program development and maintenance. If the information on inter-file dependences and command sequences is stored in a file, the simple command *make* is frequently sufficient to update the interesting files, regardless of the number that have been edited since the last "make". In most cases, the description file is easy to write and changes infrequently. It is usually easier to type the *make* command than to issue even one of the needed operations, so the typical cycle of program development operations becomes

think → edit → *make* → test . . .

Make runs on the HP-UX operating system, and is most useful for medium-sized programming projects; it does not solve the problems of maintaining multiple-source versions or of describing huge programs.

Basic Features

The basic operation of *Make* is to update a target file by ensuring that all of the files on which it depends exist and are up to date, then creating the target if it has not been modified since its dependents were. *Make* does a depth-first search of the graph of dependences. The operation of the command depends on the ability to find the date and time that a file was last modified.

To illustrate, let us consider a simple example: A program named *prog* is made by compiling and loading three C-language files *x.c*, *y.c*, and *z.c* with the *lS* library. By convention, the output of the C compilations will be found in files named *x.o*, *y.o*, and *z.o*. Assume that the files *x.c* and *y.c* share some declarations in a file named *defs*, but that *z.c* does not. That is, *x.c* and *y.c* have the line:

```
#include "defs"
```

The following text describes the relationships and operations:

```
prog :  x.o  y.o  z.o
        cc  x.o  y.o  z.o   -lS  -o  prog

x.o  y.o :   defs
```

If this information were stored in a file named *Makefile*, the command:

```
make
```

would perform the operations needed to recreate *prog* after any changes had been made to any of the four source files *x.c*, *y.c*, *z.c*, or *defs*.

Make operates using three sources of information: a user-supplied description file (as above), file names and "last-modified" times from the file system, and built-in rules to bridge some of the gaps. In our example, the first line says that *prog* depends on three "*.o*" files. Once these object files are current, the second line describes how to load them to create *prog*. The third line says that *x.o* and *y.o* depend on the file *defs*. >From the file system, *make* discovers that there are three "*.c*" files corresponding to the needed "*.o*" files, and uses built-in information on how to generate an object from a source file (*i.e.*, issue a "*cc −c*" command).

The following long-winded description file is equivalent to the one above, but takes no advantage of *make*'s innate knowledge:

```
prog :  x.o  y.o  z.o
        cc  x.o  y.o  z.o  \-lS  \-o  prog

x.o :  x.c  defs
        cc  \-c  x.c
y.o :  y.c  defs
        cc  \-c  y.c
z.o :  z.c
        cc  \-c  z.c
```

If none of the source or object files had changed since the last time *prog* was made, all of the files would be current, and the command

```
make
```

would just announce this fact and stop. If, however, the *defs* file had been edited, *x.c* and *y.c* (but not *z.c*) would be recompiled, and then *prog* would be created from the new "*.o*" files. If only the file *y.c* had changed, only it would be recompiled, but it would still be necessary to reload *prog*.

If no target name is given on the *make* command line, the first target mentioned in the description is created; otherwise the specified targets are made. The command

```
make x.o
```

would recompile *x.o* if *x.c* or *defs* had changed.

If the file exists after the commands are executed, its time of last modification is used in further decisions; otherwise the current time is used. It is often quite useful to include rules with mnemonic names and commands that do not actually produce a file with that name. These entries can take advantage of *make*'s ability to generate files and substitute macros. Thus, an entry "save" might be included to copy a certain set of files, or an entry "cleanup" might be used to throw away unneeded intermediate files. In other cases one may maintain a zero-length file purely to keep track of the time at which certain actions were performed. This technique is useful for maintaining remote archives and listings.

Make has a simple macro mechanism for substituting in dependency lines and command strings. Macros are defined by command arguments or description file lines with embedded equal signs. A macro is invoked by preceding the name by a dollar sign; macro names longer than one character must be parenthesized. The name of the macro is either the single character after the dollar sign or a name inside parentheses. The following are valid macro invocations:

```
$(CFLAGS)
$2
$(xy)
$Z
$(Z)
```

The last two invocations are identical. $$ is a dollar sign. All of these macros are assigned values during input, as shown below. Four special macros change values during the execution of the command: $*, $@, $?, and $<. They will be discussed later. The following fragment shows the use:

```
OBJECTS = x.o y.o z.o
LIBES = -lS
prog: $(OBJECTS)
        cc $(OBJECTS)  $(LIBES)  -o prog
...
```

The command

```
make
```

loads the three object files with the *lS* library. The command

```
make  "LIBES= -ll -lS"
```

loads them with both the Lex ("–ll") and the Standard ("–lS") libraries, since macro definitions on the command line override definitions in the description. (It is necessary to quote arguments with embedded blanks in HP-UX commands.)

The following sections detail the form of description files and the command line, and discuss options and built-in rules in more detail.

Description Files and Substitutions

A description file contains three types of information: macro definitions, dependency information, and executable commands. There is also a comment convention: all characters after a sharp (#) are ignored, as is the sharp itself. Blank lines and lines beginning with a sharp are totally ignored. If a non-comment line is too long, it can be continued using a backslash. If the last character of a line is a backslash, the backslash, newline, and following blanks and tabs are replaced by a single blank.

A macro definition is a line containing an equal sign not preceded by a colon or a tab. The name (string of letters and digits) to the left of the equal sign (trailing blanks and tabs are stripped) is assigned the string of characters following the equal sign (leading blanks and tabs are stripped.) The following are valid macro definitions:

```
2 = xyz
abc = -ll -ly -lS
LIBES =
```

The last definition assigns LIBES the null string. A macro that is never explicitly defined has the null string as value. Macro definitions may also appear on the *make* command line (see below).

Other lines give information about target files. The general form of an entry is:

target1 [target2 . . .] : [:] [dependent1 . . .] [; commands] [# . . .]
[*(tab)* commands] [# . . .]
. . .

Items inside brackets can be omitted. Targets and dependents are strings of letters, digits, periods, and slashes. (Shell metacharacters "*" and "?" are expanded.) A command is any string of characters not including a sharp (except in quotes) or newline. Commands may appear either after a semicolon on a dependency line or on lines beginning with a tab immediately following a dependency line.

A dependency line may have either a single or a double colon. A target name may appear on more than one dependency line, but all of those lines must be of the same (single or double colon) type.

1. For the usual single-colon case, at most one of these dependency lines may have a command sequence associated with it. If the target is out of date with any of the dependents on any of the lines, and a command sequence is specified (even a null one following a semicolon or tab), it is executed; otherwise a default creation rule may be invoked.

2. In the double-colon case, a command sequence may be associated with each dependency line; if the target is out of date with any of the files on a particular line, the associated commands are executed. A built-in rule may also be executed. This detailed form is of particular value in updating archive-type files.

If a target must be created, the sequence of commands is executed. Normally, each command line is printed and then passed to a separate invocation of the Shell after substituting for macros. (The printing is suppressed in silent mode or if the command line begins with an @ sign). *Make* normally stops if any command signals an error by returning a non-zero error code. (Errors are ignored if the "−i" flags has been specified on the *make* command line, if the fake target name ".IGNORE" appears in the description file, or if the command string in the description file begins with a hyphen. Some HP-UX commands return meaningless status). Because each command line is passed to a separate invocation of the Shell, care must be taken with certain commands (e.g., *cd* and Shell control commands) that have meaning only within a single Shell process; the results are forgotten before the next line is executed.

Before issuing any command, certain macros are set.

- $@ is set to the name of the file to be "made".
- $? is set to the string of names that were found to be younger than the target.

If the command was generated by an implicit rule (see below),

- $< is the name of the related file that caused the action, and
- $* is the prefix shared by the current and the dependent file names.

If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name ".DEFAULT" are used. If there is no such name, *make* prints a message and stops.

Command Usage

The *make* command takes four kinds of arguments: macro definitions, flags, description file names, and target file names.

```
make [ flags ] [ macro definitions ] [ targets ]
```

The following summary of the operation of the command explains how these arguments are interpreted.

First, all macro definition arguments (arguments with embedded equal signs) are analyzed and the assignments made. Command-line macros override corresponding definitions found in the description files.

Next, the flag arguments are examined. The permissible flags are

−i Ignore error codes returned by invoked commands. This mode is entered if the fake target name ".IGNORE" appears in the description file.

−s Silent mode. Do not print command lines before executing. This mode is also entered if the fake target name ".SILENT" appears in the description file.

−r Do not use the built-in rules.

−n No execute mode. Print commands, but do not execute them. Even lines beginning with an "@" sign are printed.

−t Touch the target files (causing them to be up to date) rather than issue the usual commands.

−q Question. The .IT make command returns a zero or non-zero status code depending on whether the target file is or is not up to date.

−p Print out the complete set of macro definitions and target descriptions

−d Debug mode. Print out detailed information on files and times examined.

−f Description file name. The next argument is assumed to be the name of a description file. A file name of "−" denotes the standard input. If there are no "−f" arguments, the file named *makefile* or *Makefile* in the current directory is read. The contents of the description files override the built-in rules if they are present).

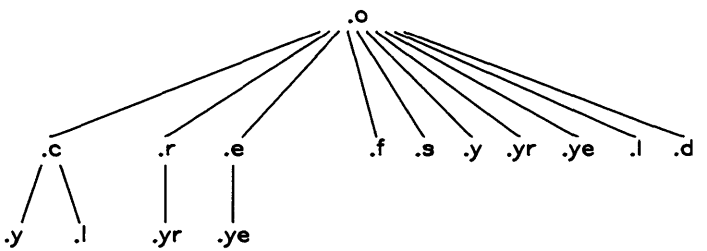
Finally, the remaining arguments are assumed to be the names of targets to be made; they are done in left to right order. If there are no such arguments, the first name in the description files that does not begin with a period is "made".

Implicit Rules

The *make* program uses a table of interesting suffixes and a set of transformation rules to supply default dependency information and implied commands. (Descriptions of these tables and means of overriding them are included at the end of this tutorial.) The default suffix list is:

.o	Object file
.c	C source file
.e	Efl source file
.r	Ratfor source file
.f	Fortran source file
.s	Assembler source file
.y	Yacc-C source grammar
.yr	Yacc-Ratfor source grammar
.ye	Yacc-Efl source grammar
.l	Lex source grammar

The following diagram summarizes the default transformation paths. If there are two paths connecting a pair of suffixes, the longer one is used only if the intermediate file exists or is named in the description.



If the file *x.o* were needed and there were an *x.c* in the description or directory, it would be compiled. If there were also an *x.l*, that grammar would be run through Lex before compiling the result. However, if there were no *x.c* but there were an *x.l*, *make* would discard the intermediate C-language file and use the direct link in the graph above.

It is possible to change the names of some of the compilers used in the default, or the flag arguments with which they are invoked by knowing the macro names used. The compiler names are the macros AS, CC, RC, EC, YACC, YACCR, YACCE, and LEX. The command

```
make CC=newcc
```

causes the "newcc" command to be used instead of the usual C compiler. The macros CFLAGS, RFLAGS, EFLAGS, YFLAGS, and LFLAGS may be set to cause these commands to be issued with optional flags. Thus,

```
make "CFLAGS=-O"
```

causes the optimizing C compiler to be used.

Example

As an example of the use of *make*, we will present the description file used to maintain the *make* command itself. The code for *make* is spread over a number of C source files and a Yacc grammar. The description file contains:

```
# Description file for the Make command

P = und -3 opr -r2          # send to GCOS to be printed
FILES = Makefile version.c defs main.c doname.c misc.c files.c dosys.c gram.y lex.c gcos.c
OBJECTS = version.o main.o doname.o misc.o files.o dosys.o gram.o
LIBES= -lS
LINT = lint -p
CFLAGS = -O

make:           $(OBJECTS)
                cc $(CFLAGS) $(OBJECTS) $(LIBES) -o make size make

$(OBJECTS):     defs
gram.o:         lex.c

cleanup:        -rm *.o gram.c
                -du

install:        @size make /usr/bin/make
                cp make /usr/bin/make ; rm make

print:          $(FILES)      # print recently changed files
                pr $? $P
                touch print

test:           make -dp grep -v TIME >1zap
                /usr/bin/make -dp grep -v TIME>2zap
                diff 1zap 2zap
                rm 1zap 2zap

lint:           dosys.c doname.c files.c main.c misc.c version.c gram.c
                $(LINT) dosys.c doname.c files.c main.c misc.c version.c gram.c
                rm gram.c

arch:           ar uv /sys/source/s2/make.a $(FILES)
```

Make usually prints out each command before issuing it. The following output results from typing the simple command

```
make
```

in a directory containing only the source and description file:

```
cc   -c version.c
cc   -c main.c
cc   -c doname.c
cc   -c misc.c
cc   -c files.c
cc   -c dosys.c
yacc   gram.y
mv y.tab.c gram.c
cc   -c gram.c
cc   version.o main.o doname.o misc.o files.o dosys.o gram.o \-lS \-o make
13188+3348+3044 = 19580b = 046174b
```

Although none of the source files or grammars were mentioned by name in the description file, *make* found them using its suffix rules and issued the needed commands. The string of digits results from the "size make" command; the printing of the command line itself was suppressed by an @ sign. The @ sign on the *size* command in the description file suppressed the printing of the command, so only the sizes are written.

The last few entries in the description file are useful maintenance sequences. The "print" entry prints only the files that have been changed since the last "make print" command. A zero-length file *print* is maintained to keep track of the time of the printing; the $? macro in the command line then picks up only the names of the files changed since *print* was touched. The printed output can be sent to a different printer or to a file by changing the definition of the *P* macro:

```
make print "P = opr -sp"
```

or

```
make print "P=  cat >zap"
```

Suggestions and Warnings

The most common difficulties arise from *make*'s specific meaning of dependency. If file *x.c* has an *#include "defs"* line, then the object file *x.o* depends on *defs*; the source file *x.c* does not. (If *defs* is changed, it is not necessary to do anything to the file *x.c*, while it is necessary to recreate *x.o*.)

To discover what *make* would do, the "−n" option is very useful. The command

```
make -n
```

orders *make* to print out the commands it would issue without actually taking the time to execute them. If a change to a file is absolutely certain to be benign (e.g., adding a new definition to an include file), the "−t" (touch) option can save a lot of time: instead of issuing a large number of superfluous recompilations, *make* updates the modification times on the affected file. Thus, the command

```
make -ts
```

("touch silently") causes the relevant files to appear up to date. Obvious care is necessary, since this mode of operation subverts the intention of *make* and destroys all memory of the previous relationships.

The debugging flag ("−d") causes *make* to print out a very detailed description of what it is doing, including the file times. The output is verbose, and recommended only as a last resort.

Appendix. Suffixes and Transformation Rules

The *make* program itself does not know what file name suffixes are interesting or how to transform a file with one suffix into a file with another suffix. This information is stored in an internal table that has the form of a description file. If the "−r" flag is used, this table is not used.

The list of suffixes is actually the dependency list for the name ".SUFFIXES"; *make* looks for a file with any of the suffixes on the list. If such a file exists, and if there is a transformation rule for that combination, *make* acts as described earlier. The transformation rule names are the concatenation of the two suffixes. The name of the rule to transform a ".*r*" file to a ".*o*" file is thus ".*r.o*". If the rule is present and no explicit command sequence has been given in the user's description files, the command sequence for the rule ".r.o" is used. If a command is generated by using one of these suffixing rules, the macro $* is given the value of the stem (everything but the suffix) of the name of the file to be made, and the macro $< is the name of the dependent that caused the action.

The order of the suffix list is significant, since it is scanned from left to right, and the first name that is formed that has both a file and a rule associated with it is used. If new names are to be appended, the user can just add an entry for ".SUFFIXES" in his own description file; the dependents will be added to the usual list. A ".SUFFIXES" line without any dependents deletes the current list. (It is necessary to clear the current list if the order of names is to be changed).

The following is an excerpt from the default rules file:

```
.SUFFIXES : .o .c .e .r .f .y .yr .ye .l .s
YACC=yacc
YACCR=yacc −r
YACCE=yacc −e
YFLAGS=
LEX=lex
LFLAGS=
CC=cc
AS=as −
CFLAGS=
RC=ec
RFLAGS=
EC=ec
EFLAGS=
FFLAGS=
.c.o :
```

```
    (CC) $(CFLAGS) -c $<
.e.o .r.o .f.o :
    (EC) $(RFLAGS) $(EFLAGS) $(FFLAGS) -c $<
.s.o :
    (AS) -o $@ $<
.y.o :
    (YACC) $(YFLAGS) $<
    (CC) $(CFLAGS) -c y.tab.c
    rm y.tab.c
    mv y.tab.o $@
.y.c :
    (YACC) $(YFLAGS) $<
    mv y.tab.c $@
```

Table of Contents

SCCS User's Guide

Introduction 1
Terms 2
S-files 2
Deltas 2
SIDs (Version Numbers) 2
ID Keywords 3
Creating SCCS Files 4
Removing SCCS Files 5
Getting Files for Compilation 6
Changing Files (Creating Deltas) 7
Getting a Copy to Edit 7
Merging the Changes Back Into the S-File 7
When To Make Deltas 8
What's Going On: The Sact Command 8
Using ID Keywords 9
Creating New Releases 11
Cancelling an Editing Session 12
Restoring Old Versions 13
Reverting to Old Versions 13
Selectively Excluding Old Deltas 14
Selectively Including Deltas 14
Removing Deltas 15
The Help Command 16
Auditing Changes 17
The Prs Command 17
Determining Why Lines Were Inserted 18
Comparing Versions 18
Files Used by SCCS 19
S-Files 19
G-Files 21
L-Files 21
P-Files 22
D-Files 22
Q-Files 23
X-Files 23
Z-Files 23
Concurrent Editing 24

Concurrent Edits on Different Versions 24
Concurrent Edits on the Same Version 25
Saving Yourself 25
Making Temporary Changes 25
Recovering an Edit File 26
Restoring the S-File 26
Using the Admin Command 27
Creating SCCS Files 27
Adding Comments to Initial Delta 28
Descriptive Text in Files 28
Setting SCCS File Flags 29
Specifying Who Can Edit a File 30
Maintaining Different Branches 32
Creating a Branch 32
Retrieving a Branch 33
Branch Numbering 33
A Warning 34
SCCS's Protection Facilities 35
General File Protection 35
System Protection Using Admin 36
Using SCCS With Make 37
To Maintain Groups of Programs 38
To Maintain a Library 39
To Maintain a Large Program 40
Using SCCS on a Multi-User Project 41
How the SCCS Interface Works 42
Configuring an SCCS System Using the Interface 42
Quick Reference 46
Commands 46
ID Keywords 48

SCCS User's Guide

Introduction

SCCS (**S**ource **C**ode **C**ontrol **S**ystem) is a set of HP-UX commands that enable you to:

- Track all changes made to a text file;
- Retrieve the current (latest) version of a file;
- Retrieve any previous version of a file, ignoring any changes made to the original after a given revision;
- Control who changes a file;
- Keep track of the date and location of each change made to a file along with the name of the person making the change;
- Add comments when each change is made indicating the reason for that change.

One application of SCCS is to keep track of source files during the development and maintenance of large systems. This article is directed towards this use of SCCS; however, it can be used in any project that involves supporting groups of related text files. Object code cannot be maintained under SCCS.

Once you store a program's source file under SCCS, all of its versions, plus additional log information, are kept in a file called the "s-file". S-files are also referred to as "SCCS files" and must have a "s." prefix on their name. There are three major operations you can perform on the s-file:

1. Get a file for some non-editing purpose, such as compilation. This operation retrieves a version of the file from the s-file that is read-only. By default, the latest version of the file is retrieved. This file is specifically NOT intended to be edited or changed in any way; any changes made to a file retrieved in this way will probably be lost.
2. Get a file for editing. This operation also retrieves a version of the file from the s-file, but this file is intended to be edited and then incorporated back into the s-file. Only one person may be editing a particular version of an s-file at a time (unless you have specifically allowed concurrent edits on the same version).
3. Merge a file back into the s-file. This is the companion operation to (2). A new version number is assigned, and comments are saved explaining why this change was made.

Terms

You need to understand several terms before using SCCS.

S-files

An s-file is a single file that holds all the different versions of your file. The s-file is stored in differential format; only the differences between versions are stored, rather than the entire text of the new version. This saves disk space and allows selective changes to be removed later. Also included in the s-file is some header information for each version, including the comments given by the person who created the version explaining why the changes were made. A description of what this header information includes is presented later in this article.

Deltas

Each set of changes to the s-file (which is approximately equivalent to a version of the file) is called a delta. Although technically a delta only includes the changes made, in practice it is usual for each delta to be made with respect to all the deltas that have occurred before. This matches normal usage, where the previous changes are not saved at all, so all changes are automatically based on all other changes that have happened through history. However, it is possible to get a version of the file that has selected deltas removed out of the middle of the list of changes. All of the deltas of a file maintained under SCCS are stored in an s-file.

SIDs (Version Numbers)

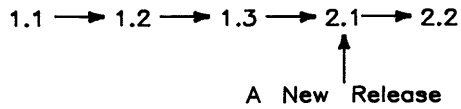
A SID (**SCCS ID**) is a number that represents a delta. This is normally a two-part number consisting of a "release" number and a "level" number. The form of two-part SIDs is:

```
release.level
```

where "release" and "level" are non-zero, positive integers. Normally the release number stays the same while the "level" increments with each delta. However, you can move into a new release of a file if some major change is being made. Since all past deltas are normally applied when version is retrieved, the SID of the final delta applied is used to represent the version number of the file as a whole.

Deltas applied to one SCCS file can be considered nodes of a tree, the initial version of the file being the root node. The root delta (node) normally has the SID number "1.1" and the deltas that follow are "1.2", "1.3", etc. The naming of successor deltas by incrementing the SID level number is performed automatically by SCCS when you retrieve a file for editing with *get -e*, although the delta itself is not created until you execute *delta*.

The following diagram illustrates the development of an SCCS file where each delta depends on all of the previous deltas.



ID Keywords

When you retrieve a version of a file from SCCS with intent to compile it (or rather, do something other than edit it), some special keywords are expanded by SCCS when they are found in the file. These ID keywords can be used to include the current version number or other information into the file. All ID keywords are of the form %x%, where "x" is an upper case letter. For example, %I% is the SID of the latest delta applied in retrieving a particular version, %W% includes the module name, SID, and a string of characters that makes it findable by the *what* command, and %G% is the date of the latest delta applied. A list of all of the ID keywords can be found in the Quick Reference section at the end of this article and in the entry for *get* in the *HP-UX Reference*.

For example, assume that you have a source file stored under SCCS and it contains the line of code:

```
static char SccsId[] = "%W%";
```

When you retrieve the file for editing, the text file will contain the line just as it appears above. However, when you retrieve the file for compilation the %W% is expanded to indicate the module name, SID, and the string of characters recognized by *what*:

```
static char SccsId[] = "@(#)prog.c     1.2     05/15/84";
```

The *what* command is a valuable tool for quickly finding out information about a particular version of a program. To use it the program's source code must be contained in SCCS files. In the SCCS files, any string of information that you want to be accessed by *what* must begin with the ID keyword %Z%. (%W%, mentioned earlier, is actually a combination of several ID keywords, including %Z%.) When the files are retrieved for compilation, this ID keyword is expanded to the string: **@(#)**. When you invoke *what* on a file, the command prints out anything it finds between this string and the first ", >, \, newline, or null character. Refer to the section "Using ID Keywords" for more information about *what*.

When you retrieve a file for editing, the ID keywords are not expanded; this is so that after you store the file back into SCCS, they can still be expanded automatically when the file is retrieved for compilation. If you edit and store a version of a file in which the ID keywords are expanded, SCCS can no longer control the updating of the ID keywords' values. For example, if you use the ID keyword for the file's version and then store the keyword's expanded value, all of the following versions will indicate that same version number – SCCS can not increment it. Also, if you compile a version of the program without expanding a version number ID keyword that appears in it, it is impossible to tell what version it is since all that the code will contain is "%I%".

Creating SCCS Files

To put source files into SCCS format, use the *admin* command. The following stores a file called "s.file" under SCCS:

```
admin  -ifile  s.file
```

The **-i** keyletter indicates that *admin* is to create a new SCCS file (called an s-file) and "initialize" its contents with the contents of the file "file". The "s.file" argument is the name of the s-file. **All s-file names must begin with "s.".** The initial version of s.file is a set of changes (delta 1.1) applied to a null s-file.

After creating a new s-file, *admin* returns the message:

```
No id keywords (cm7)
```

if you have not included any ID keywords in it. This is just a warning message and it is discussed further in a later section.

Since you have stored the contents of "s.file" under SCCS, you can now remove the original file:

```
rm  file
```

Note that if the name of the SCCS file is the same as the original text file except for the "s." prefix, then original file must be removed or moved to another directory. This is because when you retrieve a version of an SCCS file, the name of the resulting text file is the SCCS file name with the "s." removed. If there is already a writeable file with this name in your current directory, SCCS does not allow you to retrieve the SCCS file version in most cases.

Assume that your current HP-UX directory contains several C source files that you want to maintain under SCCS. The following shell script stores each under SCCS with the required "s." prefix added onto its name and removes the original source files.

```
for i in *.c
    do
        admin -i$i s.$i
        rm $i
    done
```

If you want to have ID keywords in the files, it is best to put them in before you create the s-files. If you do not, *admin* prints "No Id Keywords (cm7)" after each s-file is created. If you create an s-file without ID keywords and then later decide to add them, merely retrieve the file for editing, add the ID keywords, store the changes, and then state that ID keywords have been added when you are prompted for comments.

Removing SCCS Files

In order to protect s-files, SCCS does not supply a direct method of removing them from your system. S-files are protected from accidental deletion in two ways:

- They are created as read-only files.
- There is no SCCS command that removes them.

Because of this protection, you must make the files writeable before you can remove them. Use *chmod* to change the access permission on an s-file:

```
chmod  +w  s.file
```

The "+w" indicates that you are adding write access to the file "s.file". Once you have a writeable s-file, you can remove it with:

```
rm  s.file
```

Getting Files for Compilation

To get a copy of the latest version of the SCCS file "s.file", type:

```
get  s.file
```

Get responds, for example, with:

```
1.1
87 lines
```

indicating that version 1.1 was retrieved and that it has 87 lines. The retrieved text is placed in a file in the current directory whose name is formed by deleting the "s." prefix. The file is read-only to remind you that you are not supposed to change it. If you do make changes, they are lost the next time someone does a *get*.

To retrieve all of the SCCS files in a directory so that they can be compiled, specify the directory name as an argument to *get*:

```
get  directory
```

The retrieved text files are place in your current directory and any non-SCCS files (files without the "s." prefix) in the directory are silently ignored.

Note that if the s-file (or the directory containing s-files) that you want to access is not located in your current directory you must specify its full pathname.

Changing Files (Creating Deltas)

Getting a Copy to Edit

To edit a source file, you must first use *get* with its **-e** (**e** for **e**dit) keyletter to retrieve it:

```
get  -e  s.file
```

Get responds:

```
1.1
87 lines
New delta 1.2
```

The retrieved file "file" (without the "s." prefix) is placed in your current directory and you have read and write access to it. Edit the file using a standard text editor, for example *vi*:

```
vi  file
```

To retrieve all of the SCCS files in a directory for editing, specify the directory name as an argument to *get -e*:

```
get  -e  directory
```

Merging the Changes Back Into the S-File

When the desired changes have been made to the text file, you can store the changes in the SCCS file using the *delta* command:

```
delta  s.file
```

assuming that the s-file is located in your current directory. If it is located in a different directory you must specify a pathname for the s-file. *Delta* prompts you for "Comments?" before it merges the changes in. At this time you should type a one-line description of what the changes mean (more lines can be entered by ending each line except the last with a backslash \). *Delta* then responds, for example, with:

```
1.2
5 inserted
3 deleted
84 unchanged
```

saying that delta 1.2 was created, and it inserted five lines, removed three lines, and left 84 lines unchanged. (Changes to a line are counted as a line deleted and a line inserted.) Finally, SCCS removes "file" from your current directory; you can retrieve it again using *get*.

Note that the comments that you are prompted for are not maintained as part of the text body of the s-file, but are kept in another section of the s-file that is used internally by SCCS.

When To Make Deltas

It is probably unwise to make a delta before every recompilation or test, unless other people may need to edit the file at the same time. Creating too many deltas may result in unclear comments, such as "fixed compilation problem in previous delta" or "fixed botch in 1.3". However, it is very important to delta everything before installing a module for general use. A good technique is to edit the files you need, make all necessary changes and tests, compiling and editing as often as necessary without making deltas. When you are satisfied that you have a working version, *delta* everything being edited, re-*get* them, and recompile everything.

Working on a project with several people presents a problem when two people need to modify a particular version of a file at the same time. SCCS prevents this by locking the version while it is being edited (unless concurrent editing of one version has been specifically allowed). This means that you should not retrieve a file for editing unless you are actually going to edited it at the time, since you will be preventing other people on the project from making necessary changes. As a general rule, all source files that you are editing should be stored with *delta* before being used in compilations. This gives other users a better chance of being able to edit files when they need to.

What's Going On: The Sact Command

To find out who is currently editing an SCCS file, use:

```
sact  s.file
```

For each editing session taking place on the file, *sact* (**S**CCS **act**ivity) tells you which SID (version) is being edited, what SID will be assigned to the new delta when editing is done, who is doing the editing, and the data and time that editing began (when *get -e* was invoked). If no one is editing "s.file", *sact* returns an error message telling you that a p-file does not exist for the file (the "Types of Files" section later in this tutorial discusses p-files).

You can specify more than one SCCS file name as arguments to *sact*; each file is checked one at a time. You can also specify a directory, in which case *sact* checks every SCCS file in that directory and silently ignores non-SCCS files (files without the "s." prefix).

Using ID Keywords

ID keywords inserted into your file are expanded when you retrieve a file for compilation with *get*. They record information about the file, such as the time and date it was created, the version retrieved, and the module's name. For example, a line in an SCCS file such as:

```
static char SccsId[] = "%W%\t%G%";
```

is replaced with something like:

```
static char SccsId[] = "@(#)prog.c      1.2      08/29/80";
```

in the retrieved source file. This tells you the name and version of the source file and the time the delta was created. The string "@(#)" is the expanded form of the keyword %Z% and is searched for by the *what* command (the %W% ID keyword shown above is shorthand for several other ID keywords including %Z%). It enables you to quickly locate expanded ID keywords in text files using *what*. Note that when you retrieve a file for editing the keywords are not expanded. This is so that they will still be in their original form when you store the file again with *delta*.

Approximately 20 ID keywords are available for use in SCCS files. The "Quick Reference" section at the end of this tutorial contains a list of them, and a list can also be found under the entry for *get* in the *HP-UX Reference*.

The What Command

When %Z% is used, expanded ID keywords in files can be located using *what*. To find out the current version number of a source file and what version of it is used in an object file and final program (assuming you have previously inserted the necessary ID keywords in the SCCS source file), use:

```
what  file.c  file.o  a.out
```

What prints all strings it finds that begin with "@(#)" in the three files. It works on all types of files, including binaries and libraries. For example, the above command outputs something like:

```
file.c:
        file.c  1.2     08/29/80
file.o:
        file.c  1.1     02/05/79
a.out:
        file.c  1.1     02/05/79
```

From this you see that the source in "file.c" does not compile into the same version as the binary in "file.o" and "a.out".

What searches the given files for all occurrences of the string "@(#)", which is the replacement for the **%Z%** ID keyword, and prints what follows that string until the first double quote ("), greater than (>), blackslash (\), newline, or (nonprinting) NUL character. Note that you can locate and display constant text as well as ID keywords with *what* if you precede that text with %Z%.

For example, assume an SCCS file "s.prog.c" contains the following line:

```
char id[] "%Z%%M%:%I%;
```

Note that the colon (":") is not part of an ID keyword. It is left unchanged when the ID keywords are expanded. Next, the command line:

```
get  s.prog.c
```

is executed. The retrieved file "prog.c" is then compiled to produce "prog.o" and "a.out". The command:

```
what  prog.c  prog.o  a.out
```

produces:

```
prog.c:
    prog.c:1.2
prog.o:
    prog.c:1.2
a.out:
    prog.c:1.2
```

indicating that version 1.2 of the file "prog.c" was used in all three files.

Where to Put Id Keywords

ID keywords can be inserted anywhere in SCCS files, including comments. ID keywords that are compiled into the object module are especially useful, since they let you compare what version of the object is being run to the current version of the source.

When you put ID keywords into header files, it is important that you assign them to different variables. For example, you might use:

```
static char AccessSid[] = "%W%  %G%";
```

in the file "access.h" and:

```
static char OpsysSid[] = "%W%   %G%";
```

in the file "opsys.h". If you used the same variable name in both, you get compilation errors because the variable is redefined. You should also be aware that if you place ID keywords in a header file as code that is eventually compiled and then included that header file in several modules that are loaded together, the same version information will appear several times in the resulting object module. A solution is to insert header file's ID keywords as comments.

Creating New Releases

When you want to create a new release of a program, you can specify the new release number using *get*'s **-r** keyletter. For example:

```
get -e -r2 s.prog.c
```

retrieves the release 1's latest version of "s.prog.c" and causes the next delta to be in release 2 (an SID of 2.1). Future deltas are automatically in release 2.

To assign a new release number for all of the SCCS files in a directory, use:

```
get  -e  -r2  directory
```

assuming that the previous release was release 1, and then execute:

```
delta  directory
```

All of the SCCS files in the directory are assigned a new delta SID of 2.1

Cancelling an Editing Session

If you retrieve a file for editing with *get -e* and then decide that you do not want to edit it, cancel the editing session with:

```
unget  s.file
```

Unget returns the SID of the cancelled delta. Only the person who began an editing session can cancel it. *Unget* can accept more than one file name argument or, alternatively, use:

```
unget  -
```

in which case *unget* accepts file names from standard input.

If you are currently editing a number of SCCS files in one directory and want to cancel all of the editing sessions for them, you can specify the directory:

```
unget  directory
```

In this case *unget* checks every SCCS file in the directory. If one of the files is not currently being edited, *unget* returns an error message indicating that its associated p-file does not exist (see "Files Used by SCCS" section later in this tutorial).

If you are currently editing more than one version of a file, *unget*'s **-r** keyletter allows you to specify which version's editing session you want to cancel:

```
unget  -r2.3  s.file
```

If you find that you retrieved a file for editing when actually you needed for some other purpose, you would like to cancel the editing session but keep the file in the current directory. Normally when you cancel an editing session, *unget* removes the retrieved text file from the current directory. You can request that it not be removed with the **-n** keyletter:

```
unget  -n  s.file
```

This leaves the text file "file" still available for inspection or compilation, but any changes made to the file cannot be stored back in the SCCS file by using *delta*; and no ID keyword expansion occurs, making identification by using the *what* command impossible.

You can request that *unget* execute silently (not print out the file's cancelled delta's SID) by using the command's **-s** keyletter:

```
unget  -s  s.file
```

Restoring Old Versions

This section discusses how *get*'s **-r**, **-x**, and **-i** keyletters are used to retrieve various versions of a file. They can be used in any combination. The **-e** keyletter can also be used with them to create a new delta based on particular versions.

Reverting to Old Versions

Normally, *get* retrieves the latest version of the specified file. However, you can request a particular version using *get*'s **-r** keyletter.

Suppose that after delta 1.2 was stable you made and released a delta 1.3. However, this introduced a bug, so you made a delta 1.4 to correct it. Then you found that 1.4 was still buggy, and you decided you wanted to go back to the old version. You can access delta 1.2 by choosing the SID in a *get*:

```
get -r1.2 s.prog.c
```

This produces a version of "prog.c" that is delta 1.2. Any changes that you made between delta 1.2 and the most recent delta are ignored.

If you specify a release number but not a level number, the highest level number that exists within that release is retrieved. *Get -r* also allows you to retrieve particular branch deltas. Branches are discussed in the section "Maintaining Different Branches" later in this article.

If you try to retrieve for compilation a particular version that does not exist, SCCS responds with an error message. There is one exception: if you specify only a release number and that release doesn't exist, SCCS retrieves the delta with the highest release number that does exist, and with the highest level number within that release.

In some cases you don't know what the SID of the delta you want is. However, *get* allows you to revert to the version of the program that was running as of a certain date using its **-c** (cutoff) keyletter. For example,

```
get -c840722120000 prog.c
```

retrieves whatever version was current as of July 22, 1984 at 12:00 noon. Trailing components can be stripped off (defaulting to their highest legal value), and punctuation can be inserted in the obvious places; for example, the above line is equivalently stated with:

```
get -c"84/07/22 12:00:00" prog.c
```

Selectively Excluding Old Deltas

Suppose that you later decided that you liked the changes in delta 1.4, but that delta 1.3 should be removed. You could do this with the **-x** keyletter:

```
get -e -x1.3 s.prog.c
```

When delta 1.5 is made, it includes the changes made in delta 1.4, but excludes the changes made in delta 1.3. You can exclude a range of deltas using a dash. For example, if you don't want to include 1.3 and 1.4 you can use:

```
get -e  -x1.3-1.4 s.prog.c
```

which excludes all deltas from 1.3 to 1.4. Alternatively,

```
get -e  -x1.3-1 prog.c
```

excludes a range of deltas from 1.3 to the current highest delta in release 1.

In certain cases when using the **-x** keyletter (or **-i**, see below) there are conflicts between versions. For instance, it may be necessary to both include and delete a particular line, in which case SCCS always prints out a message telling the range of lines affected; these lines should then be examined very carefully to see if the version SCCS got is correct.

Since each delta (in the sense of "a set of changes") can be excluded at will, it is usually useful to put each semantically or conceptually distinct change into its own delta.

Selectively Including Deltas

Just as *get*'s **-x** keyletter allows you to exclude deltas from a version in which they are normally included, the **-i** allows you to include deltas that are not normally included.

For example, assume that you have an SCCS file containing five deltas, 1.1 through 1.5. To retrieve a version of a file containing only deltas 1.1, 1.3, and 1.5, request that version 1.1 be retrieved and force the inclusion of deltas 1.3 and 1.5:

```
get  -r1.1  -i1.3,1.5  s.file
```

To retrieve version 1.5 all of the deltas must be used. All of the following *get* command lines accomplish this.

```
get  -r1.5  -i1.2  s.file

get  -r1.5  s.file

get  s.file
```

Note that the **-i** keyletter in the first command line has no effect since delta 1.2 is already used to construct version 1.5. The **-r** keyletter is not required either since delta 1.5 is the most recent delta and, by default, *get* retrieves the version incorporating it.

If there are conflicts between versions when you use the **-i** keyletter, SCCS provides a message indicating the range of lines affected, just as it does when the **-x** keyletter is used. You should examine these lines in the retrieved file to make sure that they are correct.

Removing Deltas

Get -x allows you to exclude deltas from the retrieved file; however, the deltas are not removed from the SCCS file and the information they contain is still available and consuming space. To permanently remove a delta from an SCCS file, use *rmdel*. *Rmdel* requires that you use the **-r** keyletter to specify which delta is removed:

```
rmdel  -r1.3  s.file
```

Before you can use *rmdel* to remove a delta, all of the following requirements must be met:

- The specified version of the file is not currently being edited;
- the SID must be the most recent delta on its branch of the delta chain for the named file: no other deltas can depend on it;
- You originally created the delta or you are the owner of the SCCS file and the directory that it is in.

The Help Command

Error messages returned by the SCCS commands have the form:

```
ERROR : message  (code)
```

If it is not clear from "message" why the error occurred, use the associated "code" as an argument to the *help* command. Invoking:

```
help  code
```

often provides a little more explanation about the cause of the error. For example, if you execute "get program" you could receive the following message:

```
ERROR[program]: not an SCCS file (co1)
```

Executing:

```
help  co1
```

produces:

```
co1:
"not an SCCS file"
A file that you think is an SCCS file
does not begin with the characters "s.".
```

Auditing Changes

The Prs Command

When you create deltas, you presumably give reasons for the deltas in response to the "comments?" prompt. To print out these comments later, use:

```
prs s.file
```

Note that *prs* provides information about each of the deltas used to create the requested version of the file; therefore, it is a way to list the deltas upon which a particular version depends. It produces a report for each delta providing the time and date of creation, the user who created the delta, and the comments associated with the delta. For example, the output of the above command might be:

```
s.file:

D 1.3 84/04/12 08:21:35 becky 3 2  00020/00008/00021
MRs:
COMMENTS:
inserted 20 lines, removed 8 lines, 21 lines unchanged

D 1.2 84/04/11 09:21:08 becky 2 1  00008/00000/00021
MRs:
COMMENTS:
inserted 8 lines, 21 lines unchanged

D 1.1 84/04/10 06:37:14 becky 1 0  00021/00000/00000
MRs:
COMMENTS:
date and time created 84/04/10 06:37:14 by becky
```

The report indicates that the file's initial delta (created with *admin -i*) inserted 21 lines, delta 1.2 inserted 8 lines and left 21 unchanged, and delta 1.3 inserted 20 lines, removed 8 lines, and left 21 lines unchanged.

You can request information about a particular version of a file using *prs*'s **-r** keyletter:

```
prs  -r2.3  s.prog.c
```

Prs can accept multiple file names or directory names as arguments. If you request information about all of the SCCS files in a directory, you should probably redirect *prs*'s output to a file and look at it at your leisure:

```
prs  directory  >output
```

When a directory is specified, the effect is as if each SCCS file it contains were named and any non-SCCS files are ignored.

Prs also allows you to modify the information it provides using its **-d** keyletter. For example,

```
prs -d " Delta :I: Date :D:" s.prog.c
```

will list just the SCCS ID string of the file along with the date the delta was created. Refer to the *prs* entry in the *HP-UX Reference* to see how this is done.

Determining Why Lines Were Inserted

To find out why you inserted various lines in a file, you can get a copy of the file with each line preceded by the SID of the delta that created it using:

```
get -m s.prog.c
```

where the retrieved copy is called "prog.c". Once you have determined which delta inserted the line you are interested in, use *prs* to find out what that particular delta did by looking at its comment line.

Another way to find out which lines were inserted by a particular delta (e.g., 1.3) is:

```
get -m -p s.prog.c | grep '^1.3'
```

The **-p** flag causes *get* to output the retrieved text to the standard output rather than to a file.

Comparing Versions

To compare two versions of a file, use *sccsdiff*. For example,

```
sccsdiff -r1.3 -r1.6  s.prog.c
```

outputs the differences between delta 1.3 and delta 1.6 in a format similar to the format used by the *diff* command.

You can specify any number of file names with *sccsdiff* but the same two SID's specify which versions are compared for all of them. You can not specify a directory as an argument.

Files Used by SCCS

As a user of SCCS, you do not need to know all of the information covered in this section; however, it should give you a feel for the inner workings of SCCS.

There are 8 types of files that are used by SCCS and all of them are ASCII text files. They are:

S-files	SCCS files created by *admin -i*.
G-files	Text files containing the "body" of SCCS files and created by *get*.
L-files	Files containing delta dependency information and created by *get -l*.
P-files	Files created and used by SCCS to keep track of multiple edits.
D-files	Temporary files created and used by SCCS during the execution of *delta*.
Q-files	Temporary files created and used by SCCS to update p-files.
X-files	Temporary files created and used by SCCS to update s-files.
Z-files	Lock-files created and used by SCCS to prohibit simultaneous updating of s-files.

Normally, only 4 of these file types are visible to users of SCCS: s-files, g-files, l-files, and p-files. The remaining 4 types are temporary files used internally by SCCS during the execution of particular commands.

S-Files

S-files are often referred to as SCCS files in this tutorial. They contain all of the versions of files you are maintaining under SCCS. You create and name an s-file when you initially enter a file into SCCS:

```
admin  -ifile  s.file
```

"s.file" is the new s-file and "file" can now be removed. Accessing a file maintained under SCCS using SCCS commands is done using its s-file name. S-file names must begin with the prefix "s.".

The Contents of the S-File

S-files are composed of lines of ASCII text arranged in the following 6 parts:

Checksum	A line containing the "logical" sum of all the characters of the file, not including the checksum itself.
Delta Table	Information about each delta, such as type, SID, data and time of creation, and user inserted comments.
User Names	A list of login names and/or group IDs of users who are allowed to modify the file by adding or deleting deltas. You modify it using *admin*.
Flags	Indicators that control certain actions of various SCCS commands. You modify them using *admin*.
Descriptive Text	Arbitrary text provided by the user; usually a summary of the contents and purpose of the file. You modify it using *admin*.
Body	The actual text that is being administered by SCCS, intermixed with internal SCCS control lines. You modify it using *get -e* and *delta*.

You modify the Body section of the s-file whenever you create or delete deltas. You modify the User Names, Flags, and Descriptive Text sections using the *admin* command (see the "Using Admin" section later in this article). The Checksum and Delta Table are modified internally by SCCS.

Since the entire contents of an s-file is ASCII, the file can be processed with various HP-UX commands, such as *vi*, *grep*, and *cat*. This is convenient but somewhat risky in those instances in which an SCCS file must be modified manually (e.g. when the time and date of a delta were recorded incorrectly because the system clock was set incorrectly) or when you simply want to look at its contents.

NOTE

If you modify the SCCS file directly (instead of with SCCS commands), the Checksum value may be incorrect, causing you to receive an error whenever you try to retrieve a version of the file. This problem is discussed in a later section, "Restoring the S-File". You should not edit an s-file directly unless you completely understand its format.

G-Files

The *get* command creates a text file that contains a particular version of the file under SCCS control. This text file is called a **g-file** and its name is formed by removing the SCCS file's "s." prefix. It is this file that you use for inspection, compilation, or editing purposes.

G-files are created in the current directory and are owned by the real user. Their file mode depends on how *get* is invoked. If you use:

```
get  s.file
```

the resulting g-file "file" has mode 444 (read only) and is produced for inspection or compilation, but not for editing. Note that any ID keywords in the file are expanded to their appropriate values.

If you use:

```
get  -e  s.file
```

then "file" can be edited. Note that any ID keywords in the file are not expanded, allowing them to be stored back in the file when you use *delta*.

L-Files

When you retrieve an SCCS file with *get*, you can request that an **l-file** be created using the command's **-l** keyletter:

```
get  -l  s.file
```

The name of an l-file is formed by replacing the "s." prefix of the SCCS file with "l.". It contains a table indicating what deltas were used to create the retrieved version of an SCCS file. You must specifically request the creation of l-files with **-l**; by default *get* does not create them.

To send delta dependency information to standard output instead of placing it in an l-file, use:

```
get  -r2.3  -lp  s.file
```

P-Files

When you retrieve an SCCS file for editing (*get -e*), besides creating a writeable g-file containing the version's text, a **p-file** is also created. The name of a p-file is formed by replacing the "s." prefix of an SCCS file with "p.".

P-files are used internally by SCCS to keep track of multiple edits on the same SCCS file (see "Concurrent Editing"). For each edit that is in progress on a particular SCCS file (*get -e* has been executed but not the associated *delta*), the file's p-file keeps track of:

- SID of the retrieved version;
- SID that will be given to the new delta when *delta* is executed;
- Login name of the user that executed *get -e*;
- Date and time that the *get -e* was executed.

If a p-file is accidentally destroyed, it can be regenerated with:

```
get  -e  -g  s.file
```

The "-e -g" combination suppresses the retrieval of a writeable text file (g-file), but the associated p-file is created. A p-file must exist for an SCCS file before you can use *delta* on it.

When you request information with the *sact* command, you are presented with data from a p-file.

D-Files

D-files are used internally by SCCS during the execution of *delta* to hold a temporary copy of the original retrieved g-file before any editing was done. The name of a d-file is formed by replacing the "s." prefix of the associated SCCS file with "d.". When you retrieve an SCCS file for editing (*get -e*) and then invoke *delta*, SCCS creates a d-file and compares the edited g-file with the contents of the d-file to determine what has changed. These changes are then stored in the SCCS file (s-file).

When you invoke *delta*, you can request that the differences between the d-file and the g-file (the file that you retrieved and the file that you are now storing) be sent to standard output using:

```
delta  -p  s.file
```

Once *delta* is executed, you can request the same information with the *sccsdiff* command.

Q-Files

A **q-file** is a temporary copy of a p-file that is used internally by SCCS. Its name is formed by replacing the "p." prefix of the p-file with "q.". Whenever a p-file needs to be updated (because editing of a version of a file was completed with *delta* or started with *get -e*), a q-file is first created. The change is made to the q-file and then the p-file is removed and the q-file is renamed to become the new p-file. This strategy is used to ensure the integrity of the p-file in case there are any problems adding or deleting entries from the table.

X-Files

An **x-file** is a temporary copy of an s-file that is used internally by SCCS. All SCCS commands that modify an SCCS file do so by first creating and modifying an x-file. This ensures that the SCCS file is not damaged if the processing terminates abnormally. The name of this temporary copy is formed by replacing the "s." prefix of the SCCS file with "x.". When processing is complete, the old s-file is removed and the x-file is renamed to be the s-file.

Z-Files

Z-files are lock-files SCCS uses to prevent simultaneous updating of an SCCS file. They are discussed later in this article in the section "SCCS Protect Facilities".

Concurrent Editing

Concurrent Edits on Different Versions

SCCS allows different versions of one SCCS file to be edited at the same time. This means that a number of *get -e* commands can be executed on the same file provided that no two executions retrieve the same version, unless concurrent edits on the same version are allowed (see the discussion in the next section).

SCCS uses a p-file to keep track of the edits that are in progress on one file. The first execution of *get -e* causes the creation of a p-file for the specified SCCS file. Subsequent executions of the command update the p-file, adding entries in the file for each edit session that is in progress. Each entry in the p-file specifies the SID of the retrieved version, the SID that will be assigned to the new delta, and the login name of the person doing the editing. When an editing session is terminated (with *delta* or *unget*), the corresponding entry in the file's p-file is removed. If no other versions of the file are currently being edited, then the p-file itself is removed.

Before SCCS allows an editing session on a particular version of an SCCS file to begin, it makes sure that if a p-file for the file already exists there is no entry in it specifying that the version has already been retrieved. If there is no entry with that SID, SCCS adds an entry for the new editing session. If there is an entry with the same SID, SCCS generates an error message and does not allow the version to be retrieved for editing (unless multiple edits of the same version are allowed). SCCS informs you if editing is currently being done on another version of the file you request to edit.

NOTE

Multiple executions of *get -e* must be done from different directories. This is because each time any version of one file is retrieved, the resulting g-file (text file) is assigned the same name. As a result, SCCS prohibits multiple edits on the same file in the same directory because the g-file would constantly be overwritten.

In practice, multiple editing sessions are performed by different users with different working directories; therefore, this restriction normally does not cause a problem.

Concurrent Edits on the Same Version

By default, SCCS does not permit multiple executions of *get -e* on the same version of one SCCS file. Each editing session on a version begun with *get -e* must be ended with *delta* before another session can begin. However, you can change this and allow concurrent edits on the same version of a file by setting the file's **j** flag with the *admin* command (see the "Using Admin" section later in this article).

Note that if you do set a file's **j** flag, multiple editing sessions on the same version must be done in different directories, just like multiple edits on different versions.

Saving Yourself

Making Temporary Changes

If you use *get -e* to retrieve a file so that you can edit it, SCCS requires that you delta the changes that you make back into the associated s-file. Sometimes, however, it is necessary to make modifications to a file that you do not want saved.

To make temporary changes to a file possible, retrieve it from SCCS with:

```
get  s.file
```

SCCS does not expect changes to be made to the file; therefore, it gives it read-only access. You must now change the mode of the file so that you can edit it:

```
chmod +w file
```

Chmod +w adds write access to a file. Any changes that you now make to "file" cannot be stored in SCCS.

Recovering an Edit File

Sometimes you may find that you have lost a file that you were trying to edit. Unfortunately, you can't just execute *get -e* again; SCCS keeps track of the fact that someone is trying to edit that version, so it won't let you do it again. Neither can you retrieve it using *get*, since that would expand the ID keywords. Instead, you can say:

```
get -k prog.c
```

This retrieves the file and does not expand the ID keywords, so it is safe to do a delta with it.

Restoring the S-File

You may find that the SCCS file itself is corrupt. The most common way this happens is when someone edits the file directly, not through the SCCS commands. SCCS keeps a checksum that contains the "logical" sum of all of the characters in the file. If you modify the SCCS file directly the checksum may have the wrong value. No SCCS command will process a corrupted SCCS file except *admin -h* and *admin -z* as described below.

You should audit all SCCS files for corruption on a regular basis. The simplest way to do this is to execute *admin* with its **-h** keyletter on all of the SCCS file:

```
admin -h s.file1 s.file2 ...
```

or:

```
admin  -h  directory
```

This checks to see if each file's checksum is correct. The message "`corrupted file (c06)`" is produced for a file whose checksum is not correct.

If you have a corrupted SCCS file, you must first determine why its checksum is incorrect. If it is due to someone having directly modifying the file, the problem is often corrected by merely recomputing the checksum. Do this with *admin*'s **-z** keyletter:

```
admin -z prog.c
```

The checksum is recomputed to bring it into agreement with the actual contents of the file.

NOTE

Before you use *admin -z* you must find and correct the corruption problem. This is because once the checksum is recomputed, the corruption is no longer detectable. *Admin -z* does not find or fix the problem, it merely recomputes the checksum.

Using the Admin Command

Admin is used to create new SCCS files and change parameters of existing ones. When an SCCS file is created, its parameters are either initialized with keyletters or are assigned default values if no keyletters are specified.

Newly created SCCS files are given mode 444 (read-only) and are owned by the effective user. Only a user with write permission in the directory containing the SCCS file can use *admin* on it.

Creating SCCS Files

As discussed earlier, an SCCS file for a file called "prog" is created with:

```
admin  -iprog  s.prog
```

The name of the SCCS file is "s.prog". If no file name is specified with the -i keyletter, the text is read from standard input:

```
admin  -i  s.prog  <prog
```

When the SCCS file is created, the release number assigned to its initial delta is normally "1" and the level number is always "1", meaning that the first delta of the file is "1.1". You can assign a different initial release number using *admin*'s **-r** keyletter when the file is created:

```
admin  -iprog -r3  s.prog
```

Here, the initial delta is "3.1".

Adding Comments to Initial Delta

When you create an SCCS file, you can supply a comment stating the reason for the creation of the file. This is done with the **-y** keyletter:

```
admin  -ifile  -y"The reason this file was created"  s.file
```

If you do not specify an initial comment with **-y**, SCCS gives the inital delta a comment line with the form:

```
date and time created YY/MM/DD HH:MM:SS by logname
```

Descriptive Text in Files

A portion of an SCCS file is reserved for descriptive text, text that summarizes the content and purpose of the SCCS file. When you are creating an SCCS file you can insert descriptive text using *admin*'s **-t** keyletter followed by the name of a file containing the text:

```
admin  -ifile  -tdescrip  s.file
```

You can either add descriptive text to an existing SCCS file or replace the descriptive text it already contains with:

```
admin  -tnew_descrip  s.file
```

where "new_descrip" is the name of the file containing the descriptive text. To remove descriptive text from an SCCS file, use **-t** without a file name:

```
admin  -t  s.file
```

To see the descriptive text for an SCCS file, use *prs* as follows:

```
prs  -d:FD:  s.file
```

Prs's **-d** keyletter allows you to specify what information about the file that you want returned. The ":FD:" indicates that you want to see the file's descriptive text. Refer to the *HP-UX Reference* manual entry for *prs* for more information about the command's **-d** keyletter.

Setting SCCS File Flags

SCCS files have a number of parameters called **flags** that can be added and deleted using the *admin* command. These flags are maintained in a particular section of SCCS files along with their associated values where appropriate. Add flags with *admin*'s **-f** keyletter and delete them with its **-d** keyletter. For example:

```
admin -fd2.1 prog.c
```

sets the **d** flag to the value "2.1". This flag can then be deleted using:

```
admin -dd prog.c
```

The flags that you can add with *admin -f* or delete with *admin -d* are:

b Allow branches to be made using *get -e -b*.

d*SID* Default SID to be used on a *get*. If this is just a release number, the default is the highest version number for that release.

c*ceiling* Sets the highest release number for a file that can be retrieved with *get -e* to *ceiling*. *Ceiling* must be a number less than or equal to 9999. The default release ceiling for a file is 9999.

f*floor* Sets the lowest release number for a file that can be retrieved with *get -e* to *floor*. *Floor* must be a number greater than 0 and less than 9999. The default release floor for a file is 1.

i Give a fatal error during *get* or *delta* if there are no ID keywords in a file. This is useful to guarantee that a version of the file does not get merged into the s-file that has the ID keywords inserted as constants instead of internal forms.

j Allow concurrent edits on the same version (SID) of the SCCS file.

l*list* A *list* of releases that cannot be retrieved for editing (*get -e*). The *list* has the following syntax:

```
<list> ::= <range> | <list>,<range>

<range> ::= RELEASE_NUMBER | a
```

The character **a** is equivalent to specifying all of the releases for the names SCCS file. If you do not specify a *list* with the **l** flag, **a** is assumed by default.

To delete one or more "locked" releases with *admin*'s **-d** keyletter you must also use a *list* to specify which releases are to be "unlocked". For example, "admin -dla s.file" unlocks all of the releases of s.file so that they can be edited.

n Causes *delta* to create a "null" delta in each of those releases (if any) being skipped when a delta is made in a new release (e.g. in making delta 5.1 after delta 2.7, releases 3 and 4 are skipped). These null deltas serve as "anchor points" so that branch deltas may later be created from them. If this flag is not set for a file, skipped releases are non-existent in the SCCS file, preventing branch deltas from being created from them in the future.

q*text* Replace all occurrences of the ID keyword %Q% with the contents of file *text* when the SCCS file is retrieved for inspection or compilation. If the **q** flag has not been set for a file, occurrences of %Q% are not replaced with anything.

m*module* Replace all occurrences of the ID keyword %M% with the specified *module* name when the SCCS file is retrieved for inspection or compilation. If the **m** flag has not been set for a file, occurrences of %M% are replaced with the name of the SCCS file minus the "s." prefix.

t*type* Replace all occurrences of the ID keyword %Y% with the specified *type* when the SCCS file is retrieved for inspection or compilation. If the **t** flag has not been set for a file, occurrences of %Y% are not replaced with anything.

v[*pgm*] Causes *delta* to prompt for Modification Request (MR) numbers as the reason for creating a delta. If you set this flag when you create an SCCS file, *admin*'s **-m** keyletter must also be specified, even if its value is null.

You can optionally specify an MR number validation checking program called "pgm" with *admin -fvpgm*.

Specifying Who Can Edit a File

Admin's **-a** keyletter allows you to specify who can edit an SCCS file. Use it as follows:

```
admin  -alogin  s.file
```

where "login" is a user's login name or an HP-UX group ID. If it is a group ID, the effect is equivalent to specifying all login names common to that group ID. Several **-a** keyletters may be used on a single *admin* command line.

Note that *admin* can accept one or more SCCS file names or directory names as arguments. For example, the command line:

```
admin  -abill  -ajane  -ajohn  directory
```

gives HP-UX users bill, jane, and john editing priviledges to all of the SCCS files in the directory "directory". The list of users for each SCCS file in the directory is updated to show this. No one else can edit the SCCS files there unless specifically given the right with *admin -a*.

If no one has been assigned editing priviledges to a file with *admin -a*, the file's list of users is empty and anyone can edit the file (as long as they have write access to the file's directory).

A user's ability to edit an SCCS file is removed with *admin*'s **-e** keyletter. For example,

```
admin  -ebill  directory
```

removes bill from the list of users allowed to edit the SCCS files in "directory".

NOTE

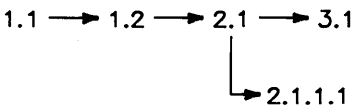
Before a user can be prohibited from editing a file, the file's list of users must be non-empty. If the list is empty everyone has editing priviledges and using *admin -e* has no effect.

When a file's list of users is non-empty, any user not added to the list with *admin -a* is already prohibited from editing the file. Thus, you can remove a specific user's editing privileges only if you have previously added him to the list of users with *admin -e*.

Maintaining Different Branches

Sometimes it is convenient to maintain an experimental version of a program for an extended period while normal maintenance continues on the version in production. This can be done using a "branch." Normally deltas continue in a straight line, each depending on the delta before. Creating a branch "forks off" a version of the program.

For example, in the diagram below there is one branch delta having an SID of 2.1.1.1:



The ability to create branches off of the latest main "trunk" delta must be enabled in advance by setting the file's **b** flag:

```
admin -fb prog.c
```

The **b** flag can also be set when the SCCS file is first created. It is not necessary to set a file's **b** flag in order to create a branch off of an older delta.

Creating a Branch

To create a branch off of the latest main trunk version, use:

```
get -e -b prog.c
```

If the retrieved version has an SID of 1.5 and no branch was previously created on it, a branch with SID 1.5.1.1 is created when the file is deltaed. The deltas for this branch are numbered 1.5.1.n where "n" increments by 1 with each delta.

If you retrieve an old version of an SCCS file for editing, SCCS automatically assigns a branch SID to the new delta. The file's **b** flag need not be set to do this. For example, assuming that the latest delta of prog.c is delta 1.5 you can create a branch off of delta 1.2 using:

```
get  -e  -r1.2  prog.c
```

SCCS will automatically number the new branch delta 1.2.1.1 if it is the first branch off delta 1.2.

Retrieving a Branch

Deltas in a branch are not normally included when you use *get*. To retrieve these versions, you have to say:

```
get -r1.5.1 prog.c
```

specifying the requested branch's SID.

Branch Numbering

SCCS uses the following SID numbering scheme for recognizing branch deltas:

```
release.level.branch.sequence
```

"Release.level" is the SID of the delta on the main trunk from which the branch descends. A "branch" number is assigned to each branch path that originates from a particular delta on the main trunk. A "sequence" number is assigned to each delta on a particular branch. Branch deltas always have all four of the above components in their SIDs and the release and level numbers are always those of the ancestral main trunk delta.

When you retrieve a branch, specifying only the release, level, and branch components of the SID returns the most recent version on a particular branch.

Although SCCS maintains enough internal information to remember delta dependencies of branch deltas, the SID number itself does not always indicate all of the deltas between a branch delta and its main trunk ancestor delta. For example, given delta 1.3.2.2 you know that the main path ancestor is delta 1.3 and that it is the second delta (sequence=2) on the second branch (branch=2) descending from delta 1.3. However, the diagrams below indicate two possible development paths for delta 1.3.2.2:

Diagram 1:

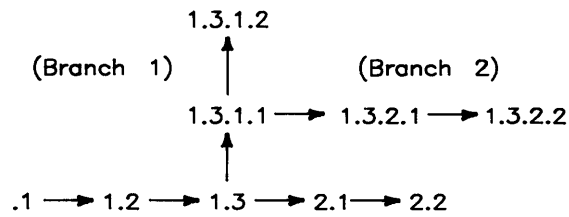
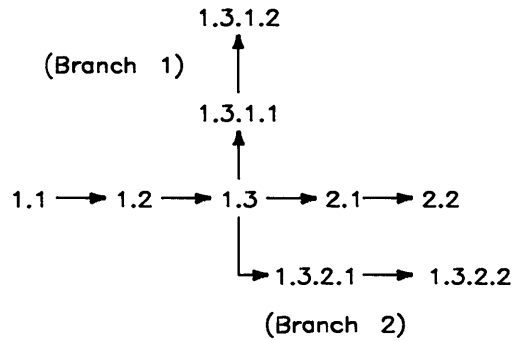


Diagram 2:



Note that in Diagram 1, version 1.3.2.2 is dependent on deltas 1.1, 1.2, 1.3, 1.3.1.1, and 1.3.2.1, while in Diagram 2 the delta with the same SID is dependent on 1.1, 1.2, 1.3, and 1.3.2.1.

A Warning

Branches should be kept to a minimum. After the first branch from the main trunk, SID's are assigned rather haphazardly, and the structure gets complex very quickly.

SCCS's Protection Facilities

The protection facilities that SCCS provides for a system fall into two categories:

- general protection of files inherent to SCCS and that incorporates general HP-UX file system protection by appropriately setting the modes of various files;
- specific system protection strategies that you control with the *admin* command.

General File Protection

New SCCS files created with *admin* are given mode 444 (read only). This mode prevents any direct modification of the files by any non-SCCS commands. The mode of the files should not be changed to allow direct modification.

SCCS files must have only one link (name) because of the way SCCS modifies the files. Commands that modify SCCS files (*delta*, *admin*) create a copy of the file. The copy, called an x-file, is modified, the original SCCS file is removed, and the copy is renamed. If the original SCCS file has any links, they are broken when it is removed. SCCS generates an error message if you try to process any file under SCCS that has multiple links.

To prevent simultaneous updates to SCCS files, when an x-file is created a **lock-file**, called the **z-file**, is also created. A z-file contains the process number of the command that creates it, and its existence is an indication to other commands that the SCCS file is being updated. Other SCCS commands that modify SCCS files will not process an SCCS file if a corresponding z-file exists. For example, assume that two people are editing two versions of an SCCS file. When one of them executes *delta*, a z-file is created which keeps the second person from successfully invoking *delta*. When *delta* has completed, the z-file is removed and the second person is free to create his own delta. Z-files are created with mode 444 (read only) in the directory containing the SCCS files and are owned by the effective user.

SCCS checks for the corruption of an SCCS file by maintaining a checksum. Whenever the file is modified with an SCCS command, its checksum is updated to reflect the logical sum of the number of characters the file has. Most SCCS commands will not allow you to access a file that is corrupted. The *admin* command allows you check for corrupted file and to correct them.

SCCS files should be kept in directories that contain only SCCS files and any temporary files created by SCCS commands. This simplifies protection and auditing of SCCS files since most of the commands allow you to operate on all of the SCCS files in a directory by merely specifying a directory name. The contents of directories should correspond to convenient logical groupings, such as subsystems of a large project.

System Protection Using Admin

Admin allows the system administrator of a project to control five major areas of protection:

1. Prohibit concurrent editing of one version of a file;
2. Specify a list of users that have permission to edit a file;
3. Prohibit editing on particular releases;
4. Set range limits to what releases users can access;
5. Make the recognition of no ID keywords in a file by SCCS commands a fatal error.

The *admin* command allows you to use these protection strategies on either a file-by-file basis or on a directory basis. How this is done is discussed in a previous section "Using Admin".

Using SCCS With Make

If you are using *make* to create and maintain systems and are using SCCS to maintain the source files for the systems, you can make the two work together by including SCCS commands in *make*'s makefiles. The following discussion assumes that you already know how to use *make*. You can refer to its entry in the *HP-UX Reference* or the article on it in *HP-UX Concepts and Tutorials* for information about it.

There are a few basic targets that most makefile should have. These are:

a.out	(or whatever the makefile generates.) This target entry regenerates whatever this makefile is supposed to regenerate. If the makefile regenerates several intermediate things, this should be called "all" and should in turn have dependencies on everything the makefile can generate.
install	Moves the objects to the final resting place, doing any special *chmod*'s as appropriate.
sources	Creates all the source files from SCCS files.
clean	Removes unneeded files from the directory.

The clean entry should not remove files that can be regenerated from the SCCS files since it is sufficiently important to have the source files around at all times.

Note that the examples of makefiles that follow are only partial and do not illustrate all of these target entries fully. Also note that the example makefiles require that you execute *make* in the same directory as the SCCS files.

To Maintain Groups of Programs

Frequently there are directories with several largely unrelated programs (such as simple commands) and these can often be maintained by one makefile. For example, the makefile below maintains "prog" and "example":

```
LDFLAGS= -i -s

prog: prog.o
        $(CC) $(LDFLAGS) -o prog prog.o
prog.o: prog.c prog.h

example: example.o
        $(CC) $(LDFLAGS) -o example example.o
example.o: example.c

.DEFAULT:
        get s.$<
```

Note that the source for the programs is maintained as SCCS files and that these files must exist in the same directory as the makefile for the makefile to be able to retrieve them. The .DEFAULT rule is called every time something is needed that does not exist, and no other rule exists to make it. The explicit dependency of the .o file on the .c file is important. Another way of doing the same thing is:

```
SRCS=   prog.c prog.h example.c

LDFLAGS= -i -s

prog: prog.o
        $(CC) $(LDFLAGS) -o prog prog.o
prog.o: prog.h

example: example.o
        $(CC) $(LDFLAGS) -o example example.o

sources: $(SRCS)
$(SRCS):
        get s.$@
```

There are some advantages to the second approach:

- the explicit dependencies of .o files on .c files are not needed;
- there is an entry called "sources" so if you just want to get all the sources you can just say "make sources";
- the makefile is less likely to do confusing things since it won't try to get things that do not exist.

To Maintain a Library

Libraries that are largely static are best updated using explicit commands, since *make* doesn't know about updating them properly. However, *make* can adequately handle libraries that are in the process of being developed. One problem in maintaining libraries is that the object (".o") files must be kept out of the library as well as in the library.

```
# configuration information
OBJS=   a.o b.o c.o d.o
SRCS=   a.c b.c c.c d.s x.h y.h z.h
TARG=   /usr/lib

# programs
GET=    get
REL=
AR=     -ar

lib.a: $(OBJS)
        $(AR) rvu lib.a $(OBJS)

install: lib.a
        cp lib.a $(TARG)/lib.a

sources: $(SRCS)
$(SRCS):
        $(GET) $(REL) s.$@

print: sources
        pr *.h *.[cs]
clean:
        rm -f *.o
        rm -f core a.out $(LIB)
```

The "$(REL)" in the $(SRCS) entry allows you to retrieve various versions of the SCCS files. For example:

```
make  REL=-r1.3
```

Note that for the install entry to execute properly, no one should be editing any of the SCCS files when it is invoked.

To Maintain a Large Program

Consisder this example makefile:

```
OBJS=   a.o b.o c.o d.o
SRCS=   a.c b.c c.y d.s x.h y.h z.h

GET=    get
REL=

a.out: $(OBJS)
        $(CC) $(LDFLAGS) $(OBJS) $(LIBS)

sources: $(SRCS)
$(SRCS):
        $(GET) $(REL) s.$@
```

(The print and clean entries are identical to the previous case.) This makefile requires copies of the source and object files to be kept during development. It is probably also wise to include lines of the form:

```
a.o: x.h y.h
b.o: z.h
c.o: x.h y.h z.h
z.h: x.h
```

so that modules are recompiled if header files change.

Since *make* does not do transitive closure on dependencies, you may find in some makefiles lines like:

```
z.h: x.h
        touch z.h
```

This would be used in cases where file z.h has a line:

```
#include "x.h"
```

in order to bring the date of z.h's last modification in line with the date of the last modification of x.h (or rather, when the system thinks z.h was last modified). Alternatively, the effect of the *touch* command can be achieved by doing a *get* on z.h.

Using SCCS on a Multi-User Project

This section describes the how SCCS is configured to maintain files for a large project that involves several users. The person that configures and controls the SCCS files is called the "SCCS System Administrator". You only need the information covered in this section if you are your project's SCCS System Administrator.

If you plan to use SCCS on a project that involves several users, you must first develop a system of controlling access to the SCCS files and commands. Thus far, this tutorial has only discussed a one-user system, where that one user has write access to the directory containing the SCCS files. The user has full use of all of the SCCS commands and can modify protected files (by first making read-only files writeable).

As an SCCS System Administrator, you should provide an interface program that gives temporary write access to the SCCS directory when users execute certain SCCS commands and and you should restrict the users to read-only access at all other times. When SCCS files used on a project, they are grouped in one directory (or more if necessary). The SCCS System Administrator is the owner of the SCCS directory, has write access to it, and has full use of all of the SCCS commands. Other users involved on the project should only have read access to the directory, which means that they can not directly use the SCCS commands that require write access.

The SCCS interface program is a C program that provides a filter for the commands requiring that the user have directory write access. If instead of using the interface program you give all of the users write access to the SCCS directory, you greatly restrict the protection facilities SCCS provides. Use of the interface provides users with only temporary write access when they execute one of the commands. The two SCCS commands that require directory write access and that must be available to the users through the interface program are *get* and *delta*. *Rmdel*, *cdc*, and *unget* also require write access and can also be made available to users through the program. The remaining SCCS commands either do not require write access to the SCCS directory or are usually used only by the SCCS System Administrator (for example, *admin*).

How the SCCS Interface Works

The SCCS interface program invokes a specified SCCS command and causes the command's process to inherit the privileges of the SCCS System Administrator for the duration of its execution. This allows the process to obtain write access to the SCCS directory.

The names of the commands that you want filtered through the interface program must be linked to the program so that invoking the command name executes the program. The interface program is written in C and when a C program is executed, the name that invoked the program is passed as argument 0 and is followed by any user-supplied arguments. By looking at the value of argument 0, the program knows which command to execute. Thus, the command name used to invoke the interface program determines which SCCS command the program executes. How other arguments, such as SCCS file names, are processed is often system dependent, but they can be passed directly to the SCCS command by the program.

Configuring an SCCS System Using the Interface

As the SCCS System Administrator, there are six basic steps that you must carry out before allowing other users to access SCCS files:

1. Create and move to an SCCS directory.
2. Write and compile the interface program.
3. Change the mode of the program.
4. Set up links between the program and the SCCS command names.
5. Modify each user's search path so that the directory containing the interface program is searched before "/usr/bin", the directory containing the SCCS commands.
6. Create the SCCS files.

Creating the SCCS Directory

Before you can successfully use the SCCS interface program, you must create one or more directories for storing the SCCS files and the program. You, as the SCCS System Administrator, should be the only one with write access to the directory.

For example, to create a directory called "/system/sccs" and then restrict write access to yourself, use:

```
mkdir  /system/sccs

chmod  755  /system/sccs
```

You must now move to the SCCS directory since you must to write and maintain the SCCS interface program there:

```
cd  /system/sccs
```

Writing and Compiling the Program

The SCCS interface program is written in C and this section assumes that you already know how to program in that language.

You should write an SCCS interface program that is customized to the needs of your system. To get you started, however, a general purpose interface program is provided below.

```
main(argc, argv)
int argc;
char *argv[];
{
 register int i;       /*counts command line arguments*/
 character cmdstr[LENGTH];    /*holds SCCS command name*/

 /*
   Do any required processing of file name arguments that
   follow the SCCS command name (arguments that don't begin
   with -)
 */

 for (i = 1; i<argc; i++)
     if (argv[i][0] != '-')
         argv[i] = filearg(argv[i]);

 /*
   Get "simple name" of name used to invoke this program
   (i.e. strip off directory-prefix name, if any).
   This step may not be needed in your system.
 */

 argv[0] = sname(argv[0]);

 /*
   Invoke actual SCCS command, passing arguments.
 */

 sprintf(cmdstr, "/usr/bin/%s", argv[0]);

 execv( cmdstr, argv);
}
```

This example program calls two routines that you must supply and that allow you to customize the SCCS interface. "Filearg" acts as a preprocessor for SCCS commands. In the program above, it is used to modify SCCS file name. This modification often involves appending the path name of an SCCS directory to the SCCS file names so that users can access the files without having to specify full path names. This routine is unnecessary if all users always specify the full pathname of the SCCS files.

The second routine that you must supply is "sname". Its purpose is to modify the name with which the user invoked the interface program so that it agrees with the name of the associated SCCS command. The statement calling this routine is not required when the link names of the interface program are the same as the names of the SCCS commands.

Once you have written an SCCS interface program designed for your system, you must compile it. Assuming that you source code file is called "interface.c", use the following to compile it:

```
cc  interface.c  -o interface
```

The name of the resulting executable program is "interface".

Specifying Program Access Permissions

The interface program must be owned by the SCCS System Administrator, and must be executable by the other users involved on the project. It must also have its "set user ID on execution" bit on so that when the program is executed, the user obtains write access to the SCCS directory. Assign these necessary characteristics to the program with:

```
chmod  4755  interface
```

where "interface" is the name of the executable interface program.

Assign Name Links to the Program

Now that you have an executable interface program, use the *cp* command to assign name links to it. It is convienent for the users if these name links are the same as the SCCS commands that are executed by the program.

To illustrate, assume that you want to allow users to access the *get* and *delta* commands through the interface program. Create the necessary links with:

```
cp  interface  get

cp  interface  delta
```

You now have three names that point to the same program: "interface", "get", and "delta". All of the other SCCS commands that require write acess to the SCCS directory will be inaccessible to the users since you have not linked them to the program.

Modifying the Users' Search Path

Once you have linked the appropriate SCCS command names to the SCCS interface program, you must modify each users' HP-UX search path so that the directory containing the the interface program is found before the actual SCCS commands. PATH is the HP-UX variable that specifies where the system looks for a command when a user executes it. When any command is executed, the system searches for the command in the directories defined by the user's PATH variable. The directories are searched in the order in which they appear in the variable's list. Your HP-UX system has a default definition for PATH but it can be redefined by each user in his *.profile* or *.login* file. Refer to your system's *HP-UX System Administrator Manual* for more information about the PATH variable and *.profile*/*.login* files.

Whether you have to change the PATH variable in every user's ".profile" file or just the system's default definition, you must insert the SCCS directory name before the appearance of "/usr/bin", the directory containing the SCCS commands, in PATH's directory list. For example, if a user's PATH variable is defined as:

```
PATH=/bin:/usr/bin
```

you should change it to:

```
PATH=/bin:/system/sccs:/usr/bin
```

where "/system/sccs" is the name of the SCCS directory containing the SCCS interface program. When you execute a command, the system first searches for it in /bin, then in /system/sccs, and finally in /usr/bin.

Creating SCCS Files

As SCCS System Administrator, you are the only user able to execute *admin* because it requires write access to the SCCS directory and you did not specify it as a link name to the SCCS interface program. Having sole access to *admin* means that you can strictly control the creation of SCCS files and the setting to their various flags. Refer back to the section "SCCS's Protection Facilities" in this tutorial for more information.

Note that in order to make full use of SCCS for a multi-user project, SCCS files should be maintained in a central location and logically grouped into one or more SCCS directories.

Quick Reference

Commands

In the discussion of the following SCCS commands, only the most useful keyletter arguments are discussed. Refer to the *HP-UX Reference* for complete descriptions of the commands and all of their keyletters.

get Gets files for compilation (not for editing). ID keywords are expanded. Note that *get -e* is listed separately. -rSID Version to get.

-p Send text to standard output rather than to the actual file.

-k Don't expand ID keywords.

-ilist List of deltas to include.

-xlist List of deltas to exclude.

-m Precede each line with SID of creating delta.

-cdate Don't apply any deltas created after date.

get -e Gets files for editing. ID keywords are not expanded. Should be matched with a *delta* command.

```
-rSID   Same as get -rSID.  If SID  specifies  a  release
        that  does  not yet exist, the highest
        numbered delta is retrieved and the new  delta
        is numbered with SID.

-b      Create a branch.

-ilist  Same as get -ilist.

-xlist  Same as get -xlist.
```

delta Merge a file retrieved with *get -e* back into the s-file. Collect comments about why this delta was made.

unget Remove a file previously retrieved with *get -e* without merging the changes into the s-file.

prs Print information about the SCCS file.

sact Determine who is currently editing a file.

what Find and print ID keywords that have been expanded. They must be preceded by @(#) (the expand form of the keyword %Z%).

admin Create or set parameters on s-files.

`-ifile`	`Create s-file, using file as the initial contents.`
`-z`	`Rebuild the checksum in case the file has been trashed.`
`-fflag[value]`	`Turn on the flag and optionally give it a value.`
`-dflag`	`Turn off (delete) the flag.`
`-tfile`	`Replace the descriptive text in the s-file with the contents of file. If file is omitted, the descriptive text is deleted from the s-file. Useful for storing documentation or "design & implementation" documents to insure they get distributed with the s-file.`
`-h`	`Check for corruption in the s-file.`

Useful flags are:

`b`	`Allow branches to be made using the -b flag to get -e.`
`dSID`	`Default SID to be used on a get.`
`i`	`Cause "No Id Keywords" error message to be a fatal error rather than a warning.`
`t`	`The module "type"; the value of this flag replaces the %Y% keyword.`

sccsdiff Compare two versions of an SCCS file.

cdc Change the comment line or MR number associated with a previously created delta.

rmdel Remove a delta from an SCCS file. This delta must be the most recent on its branch or the main trunk– no other deltas can depend on it.

help Supplies additional information about an SCCS error message.

ID Keywords

%Z%	Expands to "@(#)" for the *what* command to find. Every ID keyword string that you want *what* to see must be preceded by this keyword.
%M%	The current module name, e.g., "prog.c". Unless set by *admin*, it defaults to the file name minus the "s." prefix.
%F%	The SCCS file name.
%Y%	The value of the **t** flag as set by *admin*.
%I%	The SID of the retrieved text. The highest delta applied.
%W%	A shorthand for "%Z%%M% <tab> %I%".
%E%	The date of the delta corresponding to the "%I%" keyword (YY/MM/DD).
%G%	The date of the delta corresponding to the "%I%" keyword (MM/DD/YY).
%U%	The time the delta correspnding to the "%I%" keyword was created (HH:MM:SS).
%R%	The current release number, i.e., the first component of the "%I%" keyword.
%L%	The current level number, i.e., the second component of the "%I%" keyword.
%B%	The current branch number, i.e, the third component of the "%I%" keyword, if it exists.
%S%	The current sequence number, i.e., the fourth component of the "%I%" keyword, if it exists.
%D%	The current date (YY/MM/DD).
%H%	The current date (MM/DD/YY).
%T%	The current time (HH:MM:SS).
%Q%	The value of the **q** flag as set by *admin*.
%C%	The current line number. It is intended for identifying messages output by the program such as "this shouldn't have happened" type errors. It is *not* intended to be used on every line to provide sequence numbers.

Index

a

access permissions for sccs interface program, specifying 44
access to sccs files, admin command used to limit 30, 31
adding comments to initial delta 27
admin command 4, 26, 27, 47
admin command used to create sccs files 26
admin command used to limit access to sccs files 30, 31
admin command used to protect sccs files 36
admin command used to restore corrupt s-file 26
a.out and prog.o compiled from prog.c 10
assign new release number for all SCCS files in a directory 11
assigning name links to sccs interface program 44

b

branch numbering 33
branch numbering must be limited 34
branch retrieval 33
branches, maintaining multiple 32

c

cancelling an editing session 12
cdc command 47
changes, merging back into s-files 7
changes, temporary 25
changing files 7–12
chmod command 25
comments, adding to initial delta 27
comparing versions of a file 18
compile prog.c to form prog.o and a.out 10
concurrent edits on different versions 24
configuring an sccs system by use of sccs interface 42
creating new releases 11
creating sccs directory 42
creating SCCS files 4
creating sccs files 45

d

d-files 22
d-files used during delta execution 22
delta command 2, 7, 8, 46
deltas, including selected 14
deltas, removing 15
deltas, when to make 8
descriptive text in files 27
diff command 18
different versions, concurrent edits on 24
directory, creating sccs 42

e

edit, cancelling a session 12
edit file, lost, recovering 26
editing, concurrent, on different versions 24
editing s-files 20
editing: getting a copy 7
edits, concurrent, on same version 25
excluding selected old deltas 14

f

file access, sccs, admin command used to limit 30, 31
file, lost edit, recovering 26
file searches by what 10
file types, sccs 19
file version comparisons 18
files, descriptive text in 27
files, getting for compilation 6
files, sccs, creating 45

g

g-files, created by get command 21
g-files, editing 21
get -e command, multiple, must be executed from different directories 24
get -r command to create new release 11
get -x command 15
get command 3, 7, 10, 46
get command creates g-files 21

get command creates l-files 21
get command creates p-files 22
getting a copy to edit 7
getting files for compilation 6
groups of programs, maintaining 38

h

help command 16, 47

i

ID keyword expansion 3, 4, 10
ID keywords 3–5, 9, 47
ID keywords in header files 10
ID keywords in header files as comments 11
ID keywords placed in SCCS files 10
including selected deltas 14
interface, sccs, how it works 42

l

l-files, created by get command 21
large programs, maintaining 38
level number 2
library, maintaining 38
lost edit file, recovering 26

m

maintaining a library 38
maintaining groups of programs 38
maintaining large programs 38
maintaining multiple branches 32
make, using sccs with 37
merging changes back into s-files 7
modifying user's search path 45
multi-user project, using sccs on 41
multiple branches, maintaining 32

n

name links to sccs interface program, assigning 44
new release number assigned for all SCCS files in a directory 11
numbering branches 33

o

old deltas, selectively excluding 14
old versions, restoring or reverting to 13

p

p-file required before delta command can be used 22
p-files, created by get command 22
p-files, regenerated by get command if destroyed 22
path, search, modifying user's 45
permissions for sccs interface program access, specifying 44
print sccs delta comments 17
prog.c compiled to form prog.o and a.out 10
prog.o and a.out compiled from prog.c 10
program, sccs interface, assigning name links to 44
program, sccs interface, specifying access permissions for 44
program, sccs interface, writing and compiling 43
programs, maintaining groups of 38
programs, maintaining large 38
project, multi-user, using sccs on 41
protecting sccs files 35
prs command 17, 18, 46
prs command, used to see descriptive text in files 27

q

q-file temporary copy of p-file 23
q-file used during edit of p-file 23
q-files 23
quick reference 46–48

r

recovering lost edit file 26
release number 2
removing deltas 15
removing SCCS files 5
restoring old versions 13

restoring s-files 20, 26
retrieving a branch 33
reverting to old versions 13
rmdel command 15, 47

s

s-file 2, 4, 19
s-file contents 20
s-files, editing 20
s-files, merging changes back into 7
s-files, restoring 20, 26
sact command 8, 46
sact command presents data from p-files 22
same versions, concurrent edits on 25
sccs directory, creating 42
sccs file access, admin command used to limit 30, 31
sccs file flags, description of 28, 29
sccs file flags, setting 28
sccs file protection 35
sccs file types 19
sccs files, creating 45
SCCS ID 2
sccs interface, how it works 42
sccs interface program, assigning name links to 44
sccs interface program, specifying access permissions for 44
sccs interface program, writing and compiling 43
sccs interface, used to configure an sccs system 42
sccs sytem configured by use of sccs interface 42
sccs, use with make 37
sccs used on multi-user project 41
sccsdiff command 18, 22, 47
search path, modifying user's 45
setting sccs file flags 28
SID 2
specifying sccs interface program access permissions 44

t

temporary changes 25
text in files, descriptive 27
types of sccs files 19

u

unget command 12, 46
user search path, modifying 45
using ID keywords 8
using sccs interface to configure an sccs system 42
using sccs with make 37

v

version number 2
versions, concurrent edits on different 24
versions, concurrent edits on same 25
versions of a file, comparing 18

w

what command 3, 6, 9, 47
what, file searches by 10
when to make deltas 8
why lines inserted 18
writing and compiling sccs interface program 43

x

x-file temporary copy of s-file 23
x-file used during modifications of s-file 23
x-files 23

z

z-files 23
z-files used as lock files to protect sccs updating 23

Table of Contents

Lex: A Lexical Analyzer Generator

Introduction 1
Lex Source 4
Lex Regular Expressions 5
Operators 5
Character classes 6
Arbitrary character 7
Optional expressions 7
Repeated expressions 7
Alternation and Grouping 8
Context sensitivity 8
Repetitions and Definitions 9
Operator Precedence 9
Lex Actions 10
Example 11
Ambiguous Source Rules 14
Lex Source Definitions 16
Usage 18
HP-UX 18
Lex and Yacc 19
Examples 20
Left-Context Sensitivity 23
Character Set 26
Summary of Source Format 27
Caveats and Bugs 28

Lex: A Lexical Analyzer Generator

Introduction

Lex is a program generator designed to create C-language programs that perform lexical processing on input character streams. Lex accepts user-supplied, high-level, problem-oriented specifications for character string matching, and produces a C program that, in turn, recognizes specified regular expressions. User-written source specifications are converted by Lex into a C program that can recognize the specified regular expressions, then execute corresponding C program fragments that have also been furnished by the user.

The input character stream must consist entirely of a succession of defined, recognizable regular expressions. As the input stream is processed by the Lex-produced C program, the program executes a user-specified-and-provided C code fragment each time it encounters a recognized regular expression. If the input character stream contains an expression that is not recognized, an operation is performed on the expression that is equivalent to an *echo* command, and no other action is taken on the expression.

Lex thus provides a means for supporting a high-level-expression language where the user's freedom to write actions is unimpaired. By defining expressions and their equivalent C program fragments, the user who wishes to use a string-manipulation language for input analysis can easily convert the language to a C-based processing program without having to write processing programs in an inappropriate string-handling language.

Lex is not a complete language, but rather a generator that provides a means for adding a new language feature to the host C language. The resulting C program can then be run on any HP-UX system that provides the necessary program support hardware.

The C language is used both for the output code produced by Lex as well as the user-supplied program fragments. Compatible run-time C libraries are also provided by HP-UX, thus making Lex readily adaptable to any HP-UX-based application and hardware set, as well as to the user's background and properties of local implementations.

Lex converts the user-defined regular expressions and corresponding actions (called *source*) into the host C language. The resulting C program is stored in a file named *lex.yy.c* that contains the *yylex*() function. The *yylex* function recognizes defined expressions in an input stream (called *input*) and performs the specified actions (executes the specified C-program fragment) for each recognized expression as it is detected. See Figure 1.

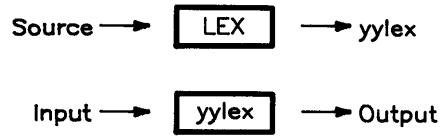


Figure 1: An overview of Lex

For a trivial example, consider a program that deletes all blanks or tabs at the ends of lines in the input character stream.

```
%%
[ \t]+$          ;              (a space is required before \t)
```

is all that is required. The program contains a %% delimiter to mark the beginning of the rules, followed by one rule. This rule contains a regular expression that matches one or more instances of the characters *blank* or *tab* (written \t for visibility, in accordance with the C language convention) just prior to the end of a line. The brackets indicate the character class made of blank and tab; the + indicates "one or more ..."; and the $ indicates "end of line," similar to ED. No action is specified, so the program generated by Lex (yylex) will ignore these characters. Everything else will be copied. To change any remaining string of blanks or tabs to a single blank, add another rule:

```
%%
[ \t]+$          ;
[ \t]+ printf(" ");
```

The finite automaton generated for this source will scan for both rules at once, observing at the termination of the string of blanks or tabs whether or not there is a newline character, and executing the desired rule action. The first rule matches all strings of blanks or tabs at the end of lines, and the second rule all remaining strings of blanks or tabs.

Lex can be used alone for simple transformations, or for analysis and statistics gathering on a lexical level. Lex can also be used with a parser generator to perform the lexical analysis phase; it is particularly easy to interface Lex and Yacc. Lex programs recognize only regular expressions; Yacc writes parsers that accept a large class of context free grammars, but require a lower level analyzer to recognize input tokens. Thus, a combination of Lex and Yacc is often appropriate. When used as a preprocessor for a later parser generator, Lex is used to partition the input stream, and the parser generator assigns structure to the resulting pieces. The flow of control in such a case (which might be the first half of a compiler, for example) is shown in Figure 2. Additional programs, written by other generators or by hand, can be added easily to programs written by Lex. Yacc users will realize that the name *yylex* is what Yacc expects its lexical analyzer to be named, so that the use of this name by Lex simplifies interfacing.

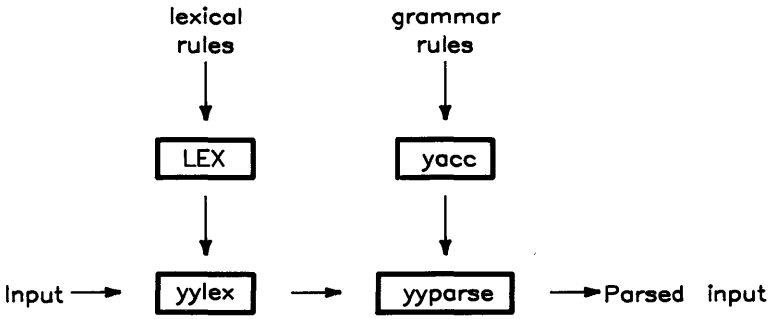


Figure 2: Lex with Yacc

Lex generates a deterministic finite automaton from the regular expressions in the source. The automaton is interpreted, rather than compiled, in order to save space. The result is still a fast analyzer. In particular, the time taken by a Lex program to recognize and partition an input stream is proportional to the length of the input. The number of Lex rules or the complexity of the rules is not important in determining speed, unless rules which include forward context require a significant amount of re~scanning. What does increase with the number and complexity of rules is the size of the finite automaton, and therefore the size of the program generated by Lex.

In the program written by Lex, the user's fragments (representing the *actions* to be performed as each regular expression is found) are gathered, as cases of a switch statement in C. The automaton interpreter directs the control flow. Opportunity is provided for the user to insert either declarations or additional statements in the routine containing the actions, or to add subroutines outside this action routine.

Lex is not limited to source which can be interpreted on the basis of one character look-ahead. For example, if there are two rules, one looking for *ab* and another for *abcdefg*, and the input stream is *abcdefh*, Lex will recognize *ab* and leave the input pointer just before *"cd. . ."* Such backup is more costly than the processing of simpler languages.

Lex Source

The general format of Lex source is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

where the definitions and the user subroutines are often omitted. The second %% is optional, but the first is required to mark the beginning of the rules. The absolute minimum Lex program is thus

```
%%
```

(no definitions, no rules) which translates into a program which copies the input to the output unchanged.

In the outline of Lex programs shown above, the *rules* represent the user's control decisions; they are a table, in which the left column contains *regular expressions* (explained in next section) and the right column contains *actions* (program fragments to be executed when the expressions are recognized). Thus an individual rule might appear

```
integer printf("found keyword INT");
```

to look for the string *integer* in the input stream and print the message "found keyword INT" whenever it appears. In this example the host procedural language is C and the C library function **printf** is used to print the string. The end of the expression is indicated by the first blank or tab character. If the action is merely a single C expression, it can just be given on the right side of the line; if it is compound, or takes more than a line, it should be enclosed in braces.

As a slightly more useful example, suppose it is desired to change a number of words from British to American spelling. Lex rules such as

```
colour      printf("color");
mechanise   printf("mechanize");
petrol      printf("gas");
```

would be a start. These rules are not quite enough, since the word *petroleum* would become *gaseum*. A way of dealing with this will be described later.

Lex Regular Expressions

The definitions of regular expressions are similar to those in ED. A regular expression specifies a set of strings to be matched. It contains text characters (which match the corresponding characters in the strings being compared) and operator characters (which specify repetitions, choices, and other features). The letters of the alphabet and the digits are always text characters; thus the regular expression

```
integer
```

matches the string *integer* wherever it appears and the expression

```
a57D
```

looks for the string *a57D*.

Operators

The operator characters are

```
" \ [ ] ^ - ? . * + | ( ) $ / { } %  < >
```

and if they are to be used as text characters, an escape should be used. The quotation mark operator (|") indicates that whatever is contained between a pair of quotes is to be taken as text characters. Thus

```
xyz"++"
```

matches the string *xyz++* when it appears. Note that a part of a string may be quoted. It is harmless but unnecessary to quote an ordinary text character; the expression

```
"xyz++"
```

is the same as the one above. Thus by quoting every non-alphanumeric character being used as a text character, the user can avoid remembering the list above of current operator characters, and is safe should further extensions to Lex lengthen the list.

An operator character may also be turned into a text character by preceding it with \ as in

```
xyz\+\+
```

which is another, less readable, equivalent of the above expressions. Another use of the quoting mechanism is to get a blank into an expression; normally, as explained above, blanks or tabs end the regular-expression portion of a rule. Any blank character not contained within [] (see below) must be quoted. Several normal C escapes with \ are recognized: \n is newline, \t is tab, and \b is backspace. To enter \ itself, use \\. Since newline is illegal in an expression, \n must be used; it is not required to escape tab and backspace. Every character but blank, tab, newline and the list above is always a text character.

Character classes

Classes of characters can be specified using the operator pair []. The construction *[abc]* matches a single character, which may be *a*, *b*, or *c*. Within square brackets, most operator meanings are ignored. Only three characters are special: these are \, – and ^. The – character indicates ranges. For example,

```
[a-z0-9<>_]
```

indicates the character class containing all the lower case letters, the digits, the angle brackets, and underline. Ranges may be given in either order. Using – between any pair of characters which are not both upper case letters, both lower case letters, or both digits is implementation dependent and will get a warning message. (E.g., [0–z] in ASCII is many more characters than it is in EBCDIC). If it is desired to include the character – in a character class, it should be first or last; thus

```
[-+0-9]
```

matches all the digits and the two signs.

In character classes, the ^ operator must appear as the first character after the left bracket; it indicates that the resulting string is to be complemented with respect to the computer character set. Thus

```
[^abc]
```

matches all characters except a, b, or c, including all special or control characters; or

```
[^a-zA-Z]
```

is any character which is not a letter. The \ character provides the usual escapes within character class brackets.

Arbitrary character

To match almost any character, the operator character

(dot or period)

is the class of all characters except newline. Escaping into octal is possible although non-portable:

```
[\40-\176]
```

matches all printable characters in the ASCII character set, from octal 40 (blank) to octal 176 (tilde).

Optional expressions

The operator "?" indicates an optional element of an expression. Thus

```
ab?c
```

matches either *ac* or *abc*.

Repeated expressions

Repetitions of classes are indicated by the operators * and +.

```
a*
```

is any number of consecutive *a* characters, including zero; while

```
a+
```

is one or more instances of *a*. For example,

```
[a-z]+
```

is all strings of lower case letters. And

```
[A-Za-z][A-Za-z0-9]*
```

indicates all alphanumeric strings with a leading alphabetic character. This is a typical expression for recognizing identifiers in computer languages.

Alternation and Grouping

The operator | indicates alternation:

```
(ab|cd)
```

matches either *ab* or *cd*. Note that parentheses are used for grouping, although they are not necessary on the outside level;

```
ab|cd
```

would have sufficed. Parentheses can be used for more complex expressions:

```
(ab|cd+)?(ef)*
```

matches such strings as *abefef*, *efefef*, *cdef*, or *cddd*; but not *abc*, *abcd*, or *abcdef*.

Context sensitivity

Lex will recognize a small amount of surrounding context. The two simplest operators for this are ^ and **$**. If the first character of an expression is ^, the expression will only be matched at the beginning of a line (after a newline character, or at the beginning of the input stream). This can never conflict with the other meaning of ^, complementation of character classes, since that only applies within the [] operators. If the very last character is **$**, the expression will only be matched at the end of a line (when immediately followed by newline). The latter operator is a special case of the / operator character, which indicates trailing context. The expression

```
ab/cd
```

matches the string *ab*, but only if followed by *cd*. Thus

```
ab$
```

is the same as

```
ab/\n
```

Left context is handled in Lex by *start conditions* as explained in the section on left context sensitivity. If a rule is only to be executed when the Lex automaton interpreter is in start condition *x*, the rule should be prefixed by

```
<x>
```

using the angle bracket operator characters. If we considered "being at the beginning of a line" to be start condition *ONE*, then the ^ operator would be equivalent to

```
<ONE>
```

Start conditions are explained more fully later.

Repetitions and Definitions

The operators **{}** specify either repetitions (if they enclose numbers) or definition expansion (if they enclose a name). For example

```
{digit}
```

looks for a predefined string named *digit* and inserts it at that point in the expression. The definitions are given in the first part of the Lex input, before the rules. In contrast,

```
a{1,5}
```

looks for 1 to 5 occurrences of *a*.

```
a{2, }
```

matches two or more occurrences of *a*, while

```
a{3}
```

matches exactly three occurences of *a* and is equivalent to *aaa*.

Finally, initial % is special, being the separator for Lex source segments.

Operator Precedence

Lex operators are handled according to the following rules of precedence:

- All operations on a single line have the same precendence.
- Lex operators are ranked in the following order of precendence, beginning with highest precedence and proceding to the lowest precedence:

```
*  ?  +
concatenation
repetition
$  ^
|
/  <>
```

Lex Actions

When an expression written as above is matched, Lex executes the corresponding action. This section describes some features of Lex which aid in writing actions. Note that there is a default action, which consists of copying the input to the output. This is performed on all strings not otherwise matched. Thus the Lex user who wishes to absorb the entire input, without producing any output, must provide rules to match everything. When Lex is being used with Yacc, this is the normal situation. One may consider that actions are what is done instead of copying the input to the output; thus, in general, a rule which merely copies can be omitted. Also, a character combination which is omitted from the rules and which appears as input is likely to be printed on the output, thus calling attention to the gap in the rules.

One of the simplest things that can be done is to ignore the input. Specifying a C null statement, ";" as an action causes this result. A frequent rule is

```
[ \t\n] ;
```

which causes the three spacing characters (blank, tab, and newline) to be ignored.

Another easy way to avoid writing actions is the action character which indicates that the action for this rule is the action for the next rule. The previous example could also have been written

```
" "             |
"\t"            |
"\n"            ;
```

with the same result, although in different style (the quotes around \n and \t are optional).

In more complex actions, the user will often want to know the actual text that matched some expression like *[a–z]+*. Lex leaves this text in an external character array named *yytext*. Thus, to print the name found, a rule like

```
[a-z]+  printf("%s", yytext);
```

will print the string in /flyytext/fR. The C function **printf** accepts a format argument and data to be printed; in this case, the format is "print string" (% indicating data conversion, and s indicating string type), and the data are the characters in *yytext*. So this just places the matched string on the output. This action is so common that it may be written as ECHO:

```
[a-z]+  ECHO;
```

is the same as the above. Since the default action is just to print the characters found, one might ask why give a rule, like this one, which merely specifies the default action? Such rules are often required to avoid matching some other rule which is not desired. For example, if there is a rule which matches *read* it will normally match the instances of *read* contained in *bread* or *readjust*; to avoid this, a rule of the form *[a–z]+* is needed. This is explained further below.

Sometimes it is more convenient to know the end of what has been found; hence Lex also provides a count *yyleng* of the number of characters matched. To count both the number of words and the number of characters in words in the input, the user might write

```
[a-zA-Z]+            {words++; chars += yyleng;}
```

which accumulates in *chars* the number of characters in the words recognized. The last character in the string matched can be accessed by

```
yytext[yyleng-1]
```

Occasionally, a Lex action may decide that a rule has not recognized the correct span of characters. Two routines are provided to aid with this situation. First, *yymore()* can be called to indicate that the next input expression recognized is to be tacked on to the end of this input. Normally, the next input string would overwrite the current entry in *yytext*. Second, *yyless (n)* may be called to indicate that not all the characters matched by the currently successful expression are wanted right now. The argument *n* indicates the number of characters in *yytext* to be retained. Further characters previously matched are returned to the input. This provides the same sort of look~ahead offered by the / operator, but in a different form.

Example

Consider a language which defines a string as a set of characters between quotation (") marks, and provides that to include a |" in a string it must be preceded by a \. The regular expression which matches that is somewhat confusing, so that it might be preferable to write

```
\"[^"]*        {
        if (yytext[yyleng-1] == '\\')
             yymore();
        else
               ... normal user processing
        }
```

which will, when faced with a string such as *"abc\"def"* first match the five characters *"abc*; then the call to *yymore()* will cause the next part of the string, *"def*, to be tacked on the end. Note that the final quote terminating the string should be picked up in the code labeled "normal processing".

The function *yyless()* might be used to reprocess text in various circumstances. Consider the C problem of distinguishing the ambiguity of "=-a". Suppose it is desired to treat this as "=- a" but print a message. A rule might be

```
=-[a-zA-Z]      {
                printf("Operator (=-) ambiguous\n");
                yyless(yyleng-1);
                ... action for =- ...
                }
```

which prints a message, returns the letter after the operator to the input stream, and treats the operator as "=-". Alternatively it might be desired to treat this as "= -a". To do this, just return the minus sign as well as the letter to the input:

```
=-[a-zA-Z]      {
                printf("Operator (=-) ambiguous\n");
                yyless(yyleng-2);
                ... action for = ...
                }
```

will perform the other interpretation. Note that the expressions for the two cases might more easily be written

```
=-/[A-Za-z]
```

in the first case and

```
=/-[A-Za-z]
```

in the second; no backup would be required in the rule action. It is not necessary to recognize the whole identifier to observe the ambiguity. The possibility of "=-3", however, makes

```
=-/[^\t\n]
```

a still better rule.

In addition to these routines, Lex also permits access to the I/O routines it uses. They are:

1. **input()** which returns the next input character;
2. **output(c)** which writes the character *c* on the output; and
3. **unput(c)** pushes the character *c* back onto the input stream to be read later by **input()**.

By default these routines are provided as macro definitions, but the user can override them and supply private versions. These routines define the relationship between external files and internal characters, and must all be retained or modified consistently. They may be redefined, to cause input or output to be transmitted to or from strange places, including other programs or internal memory; but the character set used must be consistent in all routines; a value of zero returned by *input* must mean end of file; and the relationship between *unput* and *input* must be retained or the Lex look-ahead will not work.

Lex does not look ahead at all if it does not have to, but every rule ending in +,*,?, or **$** or containing / implies look-ahead. Look-ahead is also necessary to match an expression that is a prefix of another expression. See below for a discussion of the character set used by Lex. The standard Lex library imposes a 100 character limit on backup.

Another Lex library routine that the user will sometimes want to redefine is *yywrap()* which is called whenever Lex reaches an end-of-file. If *yywrap* returns a 1, Lex continues with the normal wrapup on end of input. Sometimes, however, it is convenient to arrange for more input to arrive from a new source. In this case, the user should provide a *yywrap* which arranges for new input and returns 0. This instructs Lex to continue processing. The default *yywrap* always returns 1.

This routine is also a convenient place to print tables, summaries, etc. at the end of a program. Note that it is not possible to write a normal rule which recognizes end-of-file; the only access to this condition is through *yywrap*. In fact, unless a private version of *input()* is supplied a file containing nulls cannot be handled, since a value of 0 returned by *input* is taken to be end-of-file.

Ambiguous Source Rules

Lex can handle ambiguous specifications. When more than one expression can match the current input, Lex chooses as follows:

1. The longest match is preferred.
2. Among rules which matched the same number of characters, the rule given first is preferred.

Thus, suppose the rules

```
integer keyword action ...;
[a-z]+  identifier action ...;
```

to be given in that order. If the input is *integers*, it is taken as an identifier, because *[a–z]+* matches 8 characters while *integer* matches only 7. If the input is *integer*, both rules match 7 characters, and the keyword rule is selected because it was given first. Anything shorter (e.g. *int*) will not match the expression *integer* and so the identifier interpretation is used.

The principle of preferring the longest match makes rules containing expressions like .* dangerous. For example,

```
'.*'
```

might seem a good way of recognizing a string in single quotes. But it is an invitation for the program to read far ahead, looking for a distant single quote. Presented with the input

```
'first' quoted string here, 'second' here
```

the above expression will match

```
'first' quoted string here, 'second'
```

which is probably not what was wanted. A better rule is of the form

```
'[^'\n]*'
```

which, on the above input, will stop after *'first'*. The consequences of errors like this are mitigated by the fact that the . operator will not match newline. Thus expressions like .* stop on the current line. Don't try to defeat this with expressions like *[.\n]+* or equivalents; the Lex generated program will try to read the entire input file, causing internal buffer overflows.

Note that Lex is normally partitioning the input stream, not searching for all possible matches of each expression. This means that each character is accounted for once and only once. For example, suppose it is desired to count occurrences of both *she* and *he* in an input text. Some Lex rules to do this might be

```
she     s++;
he      h++;
\n      |
.       ;
```

where the last two rules ignore everything besides *he* and *she*. Remember that . does not include newline. Since *she* includes *he*, Lex will normally *not* recognize the instances of *he* included in *she*, since once it has passed a *she* those characters are gone.

Sometimes the user would like to override this choice. The action REJECT means "go do the next alternative." It causes whatever rule was second choice after the current rule to be executed. The position of the input pointer is adjusted accordingly. Suppose the user really wants to count the included instances of *he*:

```
she     {s++; REJECT;}
he      {h++; REJECT;}
\n      |
.       ;
```

these rules are one way of changing the previous example to do just that. After counting each expression, it is rejected; whenever appropriate, the other expression will then be counted. In this example, of course, the user could note that *she* includes *he* but not vice versa, and omit the REJECT action on *he*; in other cases, however, it would not be possible *a priori* to tell which input characters were in both classes.

Consider the two rules

```
a[bc]+  { ... ; REJECT;}
a[cd]+  { ... ; REJECT;}
```

If the input is *ab*, only the first rule matches, and on *ad* only the second matches. The input string *accb* matches the first rule for four characters and then the second rule for three characters. In contrast, the input *accd* agrees with the second rule for four characters and then the first rule for three.

In general, REJECT is useful whenever the purpose of Lex is not to partition the input stream but to detect all examples of some items in the input, and the instances of these items may overlap or include each other. Suppose a digram table of the input is desired; normally the digrams overlap, that is the word *the* is considered to contain both *th* and *he*. Assuming a two-dimensional array named *digram* to be incremented, the appropriate source is

```
%%
        [a-z][a-z]      {digram[yytext[0]][yytext[1]]++; REJECT;}
        .       ;
        \n      ;
```

where the REJECT is necessary to pick up a letter pair beginning at every character, rather than at every other character.

Lex Source Definitions

Remember the format of the Lex source:

```
{definitions}
%%
{rules}
%%
{user routines}
```

So far only the rules have been described. The user needs additional options, though, to define variables for use in his program and for use by Lex. These can go either in the definitions section or in the rules section.

Remember that Lex is turning the rules into a program. Any source not intercepted by Lex is copied into the generated program. There are three classes of such things.

1. Any line which is not part of a Lex rule or action which begins with a blank or tab is copied into the Lex generated program. Such source input prior to the first %% delimiter will be external to any function in the code; if it appears immediately after the first %%, it appears in an appropriate place for declarations in the function written by Lex which contains the actions. This material must look like program fragments, and should precede the first Lex rule.

 As a side effect of the above, lines which begin with a blank or tab, and which contain a comment, are passed through to the generated program. This can be used to include comments in either the Lex source or the generated code. The comments should follow the host language convention.

2. Anything included between lines containing only %{ and %} is copied out as above. The delimiters are discarded. This format permits entering text like preprocessor statements that must begin in column 1, or copying lines that do not look like programs.
3. Anything after the third %% delimiter, regardless of formats, etc., is copied out after the Lex output.

Definitions intended for Lex are given before the first %% delimiter. Any line in this section not contained between %{ and %}, and begining in column 1, is assumed to define Lex substitution strings. The format of such lines is

```
name translation
```

and it causes the string given as a translation to be associated with the name. The name and translation must be separated by at least one blank or tab, and the name must begin with a letter. The translation can then be called out by the {name} syntax in a rule. Using {D} for the digits and {E} for an exponent field, for example, might abbreviate rules to recognize numbers:

```
D                       [0-9]
E                       [DEde][-+]?{D}+
%%
{D}+                    printf("integer");
{D}+"."{D}*({E})?       |
{D}*"."{D}+({E})?       |
{D}+{E}                 printf("real");
```

Note the first two rules for real numbers; both require a decimal point and contain an optional exponent field, but the first requires at least one digit before the decimal point and the second requires at least one digit after the decimal point. To correctly handle the problem posed by a Fortran expression such as *35.EQ.I*, which does not contain a real number, a context-sensitive rule such as

```
[0-9]+/"."EQ     printf("integer");
```

could be used in addition to the normal rule for integers.

The definitions section may also contain other commands, including the selection of a host language, a character set table, a list of start conditions, or adjustments to the default size of arrays within Lex itself for larger source programs. These possibilities are discussed later in the section "Summary of Source Format,".

Usage

There are two steps in compiling a Lex source program. First, the Lex source must be turned into a generated program in the host general purpose language. Then this program must be compiled and loaded, usually with a library of Lex subroutines. The generated program is on a file named *lex.yy.c*. The I/O library is defined in terms of the C standard library.

HP-UX

The library is accessed by the loader flag *—ll* for C, so an appropriate set of commands is

```
lex source
cc lex.yy.c -ll
```

The resulting program is placed on the usual file *a.out* for later execution. To use Lex with Yacc see below. Although the default Lex I/O routines use the C standard library, the Lex automata themselves do not do so; if private versions of *input, output* and *unput* are given, the library can be avoided.

Lex and Yacc

If you want to use Lex with Yacc, note that what Lex writes is a program named *yylex()*, the name required by Yacc for its analyzer. Normally, the default main program on the Lex library calls this routine, but if Yacc is loaded, and its main program is used, Yacc will call *yylex()*. In this case each Lex rule should end with

```
return(token);
```

where the appropriate token value is returned. An easy way to get access to Yacc's names for tokens is to compile the Lex output file as part of the Yacc output file by placing the line

```
# include "lex.yy.c"
```

in the last section of Yacc input. Supposing the grammar to be named "good" and the lexical rules to be named "better" the HP-UX command sequence can just be:

```
yacc good
lex better
cc y.tab.c -ly -ll
```

The Yacc library (−ly) should be loaded before the Lex library, to obtain a main program which invokes the Yacc parser. The generations of Lex and Yacc programs can be done in either order.

Alternatively, the **-d** option of *yacc* can be used to generate a file *y.tab.h* of token definitions. This can be included in the Lex program by placing

```
%{
#include "y.tab.h"
%}
```

in the definitions section of the Lex input file. If the grammar is *gram.y* and the lexical rules are in file *scan.l*, the HP-UX command sequence is:

```
yacc -d gram.y
lex scan.l
cc y.tab.c lex.yy.c -ly -ll
```

Examples

As a simple example, consider copying an input file while adding 3 to every positive number which is divisible by 7. Here is a suitable Lex source program

```
%%
        int k;
[0-9]+  {
        k = atoi(yytext);
        if (k%7 == 0)
             printf("%d", k+3);
        else
             printf("%d",k);
        }
```

to do just that. The rule [0—9]+ recognizes strings of digits; *atoi* converts the digits to binary and stores the result in *k*. The operator % (remainder) is used to check whether *k* is divisible by 7; if it is, it is incremented by 3 as it is written out. It may be objected that this program will alter such input items as *49.63* or *X7*. Furthermore, it increments the absolute value of all negative numbers divisible by 7. To avoid this, just add a few more rules after the active one, as here:

```
%%
        int k;
-?[0-9]+                {
                        k = atoi(yytext);
                        printf("%d", (k%7 == 0)&&(k>0)? k+3 : k);
                        }
-?[0-9.]+               ECHO;
[A-Za-z][A-Za-z0-9]+    ECHO;
```

Numerical strings containing a "." or preceded by a letter will be picked up by one of the last two rules, and not changed. The *if-else* has been replaced by a C conditional expression to save space; the form *a?b:c* means "if *a* then *b* else *c*."

For an example of statistics gathering, here is a program which histograms the lengths of words, where a word is defined as a string of letters.

```
    int lengs[100];
%%
    [a-z]+  lengs[yyleng]++;
    .       |
    \n      ;
    %%
    yywrap()
    {
    int i;
    printf("Length  No. words\n");
    for(i=0; i<100; i++)
         if (lengs[i] > 0)
              printf("%5d%10d\n",i,lengs[i]);
    return(1);
    }
```

This program accumulates the histogram, while producing no output. At the end of the input it prints the table. The final statement *return(1)*; indicates that Lex is to perform wrapup. If *yywrap* returns zero (false) it implies that further input is available and the program is to continue reading and processing. To provide a *yywrap* that never returns true causes an infinite loop.

As a larger example, here are some program fragments which converts double precision Fortran to single precision Fortran. Because Fortran does not distinguish upper and lower case letters, this routine begins by defining a set of classes including both cases of each letter:

```
a       [aA]
b       [bB]
c       [cC]
...
z       [zZ]
```

An additional class recognizes white space:

```
W       [ \t]*
```

The first rule changes "double precision" to "real", or "DOUBLE PRECISION" to "REAL".

```
{d}{o}{u}{b}{l}{e}{W}{p}{r}{e}{c}{i}{s}{i}{o}{n} {
     printf(yytext[0]=='d'? "real" : "REAL");
     }
```

Care is taken throughout this program to preserve the case (upper or lower) of the original program. The conditional operator is used to select the proper form of the keyword. The next rule copies continuation card indications to avoid confusing them with constants:

```
^"      "[^ 0]    ECHO;
```

In the regular expression, the quotes surround the blanks. It is interpreted as "beginning of line, then five blanks, then anything but blank or zero." Note the two different meanings of ^. There follow some rules to change double precision constants to ordinary floating constants.

```
[0-9]+{W}{d}{W}[+-]?{W}[0-9]+          |
[0-9]+{W}"."{W}{d}{W}[+-]?{W}[0-9]+    |
"."{W}[0-9]+{W}{d}{W}[+-]?{W}[0-9]+    {
     /* convert constants */
     for(p=yytext; *p != 0; p++)
          {
          if (*p == 'd' | *p == 'D')
               *p=+ 'e'- 'd';
          ECHO;
          }
```

After the floating point constant is recognized, it is scanned by the *for* loop to find the letter *d* or *D*. The program than adds *'e'—'d'* which converts it to the next letter of the alphabet. The modified constant, now single-precision, is written out again. There follow a series of names which must be respelled to remove their initial *d*. By using the array *yytext* the same action suffices for all the names (only a sample of a rather long list is given here).

```
{d}{s}{i}{n}     |
{d}{c}{o}{s}     |
{d}{s}{q}{r}{t}  |
{d}{a}{t}{a}{n}  |
...
{d}{f}{l}{o}{a}{t}       printf("%s",yytext+1);
```

Another list of names must have initial *d* changed to initial *a*:

```
{d}{l}{o}{g}     |
{d}{l}{o}{g}10   |
{d}{m}{i}{n}1    |
{d}{m}{a}{x}1    {
                 yytext[0] =+ 'a' - 'd';
                 ECHO;
                 }
```

And one routine must have initial *d* changed to initial *r*:

```
{d}1{m}{a}{c}{h}        {yytext[0] =+ 'r'  - 'd';
                        ECHO;
                        }
```

To avoid such names as *dsinx* being detected as instances of *dsin*, some final rules pick up longer words as identifiers and copy some surviving characters:

```
[A-Za-z][A-Za-z0-9]*    |
[0-9]+                  |
\n                      |
                        ECHO;
```

Note that this program is not complete; it does not deal with the spacing problems in Fortran or with the use of keywords as identifiers.

Left-Context Sensitivity

Sometimes it is desirable to have several sets of lexical rules to be applied at different times in the input. For example, a compiler preprocessor might distinguish preprocessor statements and analyze them differently from ordinary statements. This requires sensitivity to prior context, and there are several ways of handling such problems. The ^ operator, for example, is a prior context operator, recognizing immediately preceding left context just as $ recognizes immediately following right context. Adjacent left context could be extended, to produce a facility similar to that for adjacent right context, but it is unlikely to be as useful, since often the relevant left context appeared some time earlier, such as at the beginning of a line.

This section describes two means of dealing with different environments: a simple use of flags, when only a few rules change from one environment to another and the use of *start conditions* on rules run together. In each case, there are rules which recognize the need to change the environment in which the succeeding input text is analyzed, and set some parameter to reflect the change. This may be a flag explicitly tested by the user's action code; such a flag is the simplest way of dealing with the problem, since Lex is not involved at all. It may be more convenient, however, to have Lex remember the flags as initial conditions on the rules. Any rule may be associated with a start condition. It will only be recognized when Lex is in that start condition. The current start condition may be changed at any time. Finally, if the sets of rules for the different environments are very dissimilar, clarity may be best achieved by writing several distinct lexical analyzers, and switching from one to another as desired.

Consider the following problem: copy the input to the output, changing the word *magic* to *first* on every line which began with the letter *a*, changing *magic* to *second* on every line which began with the letter *b*, and changing *magic* to *third* on every line which began with the letter *c*. All other words and all other lines are left unchanged.

These rules are so simple that the easiest way to do this job is with a flag:

```
        int flag;
%%
^a      {flag = 'a'; ECHO;}
^b      {flag = 'b'; ECHO;}
^c      {flag = 'c'; ECHO;}
\n      {flag =  0 ; ECHO;}
magic   {
        switch (flag)
        {
        case 'a': printf("first"); break;
        case 'b': printf("second"); break;
    case 'c': printf("third"); break;
        default: ECHO; break;
        }
        }
```

should be adequate.

To handle the same problem with start conditions, each start condition must be introduced to Lex in the definitions section with a line reading

```
%Start  name1 name2 ...
```

where the conditions may be named in any order. The word *Start* may be abbreviated to *s* or *S*. The conditions may be referenced at the head of a rule with the <> brackets:

```
<name1>expression
```

is a rule which is only recognized when Lex is in the start condition *name1*. To enter a start condition, execute the action statement

```
BEGIN name1;
```

which changes the start condition to *name1*. To resume the normal state,

```
BEGIN 0;
```

or

```
BEGIN INITIAL
```

resets the initial condition of the Lex automaton interpreter.

A rule may be active in several start conditions:

```
<name1,name2,name3>
```

is a legal prefix. Any rule not beginning with the <> prefix operator is always active. To specify that a rule is active **only** in the normal state, prefix it with <INITIAL>. Note that "INITIAL" is predefined by Lex, and should not be included in a **%start** declaration.

The same example as before can be written:

```
%START AA BB CC
%%
^a      {ECHO; BEGIN AA;}
^b      {ECHO; BEGIN BB;}
^c      {ECHO; BEGIN CC;}
\n      {ECHO; BEGIN 0;}
<AA>magic       printf("first");
<BB>magic       printf("second");
<CC>magic       printf("third");
```

where the logic is exactly the same as in the previous method of handling the problem, but Lex does the work rather than the user's code.

Character Set

The programs generated by Lex handle character I/O only through the routines *input, output,* and *unput*. Thus the character representation provided in these routines is accepted by Lex and employed to return values in *yytext*. For internal use, a character is represented as a small integer which, if the standard library is used, has a value equal to the integer value of the bit pattern representing the character on the host computer. Normally, the letter *a* is represented as the same form as the character constant *'a'*. If this interpretation is changed, by providing I/O routines which translate the characters, Lex must be told about it, by giving a translation table. This table must be in the definitions section, and must be bracketed by lines containing only "%T". The table contains lines of the form

```
{integer} {character string}
```

which indicate the value associated with each character. Thus the next example maps the lower and upper case letters together into the integers 1 through 26, newline into 27, + and − into 28 and 29, and the digits into 30 through 39. Note the escape for newline. If a table is supplied, every character that is to appear either in the rules or in any valid input must be included in the table. No character may be assigned the number 0, and no character may be assigned a bigger number than the size of the hardware character set.

```
%T
 1      Aa
 2      Bb
...
26      Zz
27      \n
28      +
29      -
30      0
31      1
...
39      9
%T
```

Sample character table.

Summary of Source Format

The general form of a Lex source file is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

The definitions section contains a combination of

1. Definitions, in the form "name space translation".
2. Included code, in the form "space code".
3. Included code, in the form

   ```
   %{
   code
   %}
   ```

4. Start conditions, given in the form

   ```
   %S name1 name2 ...
   ```

5. Character set tables, in the form

   ```
   %T
   number space character-string
   ...
   %T
   ```

6. Changes to internal array sizes, in the form

 %x *nnn*

 where *nnn* is a decimal integer representing an array size and *x* selects the parameter as follows:

Letter	*Parameter*
p	positions
n	states
e	tree nodes
a	transitions
k	packed character classes
o	output array size

Lines in the rules section have the form "expression action" where the action may be continued on succeeding lines by using braces to delimit it.

Regular expressions in Lex use the following operators:

```
x       the character "x"
"x"     an "x", even if x is an operator.
\x      an "x", even if x is an operator.
[xy]    the character x or y.
[x-z]   the characters x, y or z.
[^x]    any character but x.
.       any character but newline.
^x      an x at the beginning of a line.
<y>x    an x when Lex is in start condition y.
x$      an x at the end of a line.
x?      an optional x.
x*      0,1,2, ... instances of x.
x+      1,2,3, ... instances of x.
x|y     an x or a y.
(x)     an x.
x/y     an x but only if followed by y.
{xx}    the translation of xx from the definitions section.
x{m,n}  m through n occurrences of x
```

Caveats and Bugs

There are pathological expressions which produce exponential growth of the tables when converted to deterministic machines; fortunately, they are rare.

REJECT does not rescan the input; instead it remembers the results of the previous scan. This means that if a rule with trailing context is found, and REJECT executed, the user must not have used *unput* to change the characters forthcoming from the input stream. This is the only restriction on the user's ability to manipulate the not-yet-processed input.

Index

a

action execution by *lex* 10
alternation operator 8
ambiguous source rultes 14
arbitrary-character matching (dot) 7

b

bugs 28

c

character classes 6
character I/O 26
compiling *lex* source programs 18
context handling 8

d

definition expansion 9

e

expressions, optional 7
expressions, repeated 7

g

grouping characters 8

h

HP-UX usage 18

i

ignore input 10
I/O 26

l

left-context sensitivity 23
lex source definitions 16
lex used with *yacc* 2, 19
look-ahead 11
look-ahead, implied 13

m

matched expression retrieval 10

n

numeric repetitions 9

o

operator characters 5
operator precedence 9
optional expressions 7

p

precedence, operator 9
prior context sensitivity 23

r

regular expressions 5
REJECT 15
repeated expressions 7

s

source format 4
source format summary 27
source rules definitions 16

y

yacc used with *lex* 2, 19

MANUAL COMMENT CARD

Programming Environment
HP-UX Concepts and Tutorials

HP Part Number 97089-90042 August 1986

Please help us improve this manual. Circle the numbers in the following statement that best indicate how useful you found this manual. Then add any further comments in the spaces below. **In appreciation of your time, we will enter your name in a quarterly drawing for an HP calculator.** Thank you.

The information in this manual:

Is poorly organized	1	2	3	4	5	Is well organized
Is hard to find	1	2	3	4	5	Is easy to find
Doesn't cover enough	1	2	3	4	5	Covers everything
Has too many errors	1	2	3	4	5	Is very accurate

Particular pages with errors? _____

Comments: _____

Name: _____

Job Title: _____

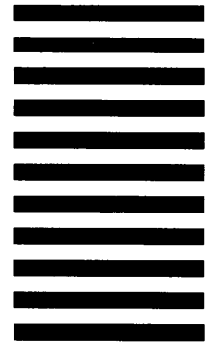
Company: _____

Address: _____

☐ Check here if you wish a reply.



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 37 LOVELAND, COLORADO

POSTAGE WILL BE PAID BY ADDRESSEE

Hewlett-Packard Company
Fort Collins Systems Division
Attn: Customer Documentation
3404 East Harmony Road
Fort Collins, Colorado 80525

MANUAL COMMENT CARD

Programming Environment
HP-UX Concepts and Tutorials

HP Part Number 97089-90042 August 1986

Please help us improve this manual. Circle the numbers in the following statement that best indicate how useful you found this manual. Then add any further comments in the spaces below. **In appreciation of your time, we will enter your name in a quarterly drawing for an HP calculator.** Thank you.

The information in this manual:

Is poorly organized	1	2	3	4	5	Is well organized
Is hard to find	1	2	3	4	5	Is easy to find
Doesn't cover enough	1	2	3	4	5	Covers everything
Has too many errors	1	2	3	4	5	Is very accurate

Particular pages with errors? _____

Comments: _____

Name: _____

Job Title: _____

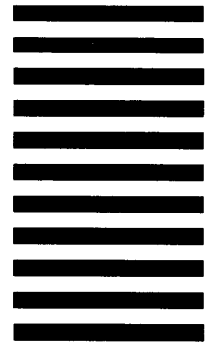
Company: _____

Address: _____

☐ Check here if you wish a reply.



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL

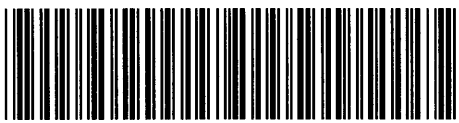
FIRST CLASS PERMIT NO. 37 LOVELAND, COLORADO

POSTAGE WILL BE PAID BY ADDRESSEE

Hewlett-Packard Company
Fort Collins Systems Division
Attn: Customer Documentation
3404 East Harmony Road
Fort Collins, Colorado 80525



HP Part Number
97089-90042
Microfiche No. 97089-99042
Printed in U.S.A. 8/86



For Internal Use Only